A SIMPLICIAL ALGORITHM
FOR CONCAVE PROGRAMMING

Robert B. Wilson

A thesis submitted in partial fulfillment of
the requirements for the degree of
Doctor of Business Administration

Graduate School of Business Administration
George F. Baker Foundation
Harvard University
June, 1963
Abstract

A SIMPLICIAL ALGORITHM FOR CONCAVE PROGRAMMING

Robert B. Wilson

Mathematical programming is a quantitative technique that has been gaining widespread acceptance in business practice as an analytical aid in complicated decision problems. The decision problem, for example, might be to specify a program of production operations and inventory stocking levels for a multi-product, multi-plant firm which must cope with capacity limitations, and which must decide what to produce, and how, before market demands can be known. In another type of problem an allocation of effort must be made among activities using fixed and variable factors of production in different amounts. In substance, the technique of mathematical programming consists of using mathematical methods to find a program which yields the maximum feasible value of some index of performance.

This thesis is a contribution to the body of programming theory and methods. The main result is a computational method for solving problems satisfying certain weak restrictions. Although the Simplex Method for the special case of linear programming has been widely used for a decade, general methods have been developed only recently. The method proposed here extends the rationale of the Simplex Method to more general problems.

The new method has been implemented in computer routines for solving three special types of programming problems: quadratic programming, linear programming under uncertainty, and variable-factor programming. These routines are available for general use, and complete descriptions are included in the thesis along with instructions for using them.
Preface

The research reported in this thesis stems from my investigations in 1961 with Professor John Bishop, into methods of solving a problem of linear programming under uncertainty (cf. Section 5). The problem had been introduced to me by my long time mentor, Professor Howard Raiffa. Although originally proposed as a topic in statistical decision theory (my field of major interest) the problem became one of developing computational methods for mathematical programming: the terminal decision problem, let alone the sampling problem, was far from solved.

Following the lead of Professor Bishop's formulation and structural analysis of the problem, I presented a paper to his seminar on the topic in the fall of 1961. That paper contained in a crude form the computational method which in its matured form is the subject of this thesis. In a revised form the paper was delivered at the Chicago Symposium on Mathematical Programming in June, 1962. Since then, the method has been generalized (and also simplified for exposition), and implemented in computer routines.

Professors Bishop (chairman), Raiffa, and Robert Dorfman have served as the supervising committee for the thesis. I am deeply indebted to each of these men for his efforts in reviewing and helping me with my work. Professor Raiffa deserves special thanks for forcing me to clarify and simplify my ideas in the exposition, and for valiantly trying to teach me how to write. It is a certain fact that the strengths of this thesis are a reflection of these men, and that the weaknesses are my own, since any one of them could, without effort, have written this thesis better than I have.

I would like to add that Drs. Philip Wolfe and Albert Madansky of the RAND Corporation contributed substantially at a crucial point in the development of my ideas by offering a reformulation of the method which bypasses the duality theory of programming.

The Ford Foundation provided financial support for my doctoral studies, for which I am very grateful. It is not, however, responsible
for the views expressed in the thesis. Also, the Division of Research of Harvard Business School made it possible for me to attend the Chicago Symposium.
Table of Contents

<table>
<thead>
<tr>
<th>Sections</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>A Simplicial Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Quadratic Programming</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Concave Programming</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>A Linear Programming Problem Under Uncertainty</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>Variable-Factor Programming</td>
<td>58</td>
</tr>
</tbody>
</table>

Appendices

A  Solution of a Numerical Example by the Simplicial Algorithm

B  A Numerical Example of Quadratic Programming Solved by the Simplicial Algorithm

C  The Computer Routines

Bibliography
1. Introduction

This paper reports my research on a method of solving mathematical programming problems. Although my investigations covered studies in programming theory as well as the development of new computational procedures, my main concern here is the computational work. I will describe a programming method, which I have developed in the course of my research, demonstrate its application on sample problems, and show how it is implemented in computer routines.

The research was motivated by a need to find a general algorithm for solving the constrained extrema problems of mathematical programming. This need arises primarily from industrial problems, but extremal problems are also important in managerial economics. In both the applied and theoretical contexts, a feasible and efficient computational procedure is necessary in order to determine specific, quantified recommendations for maximizing behavior. In addition, an algorithm can be used to explore structural relationships in the decision environment, and to do sensitivity analyses to find the relative magnitude of environmental effects. Even if a decision maker does not use its full potential, a practical computational procedure is an essential companion to normative prescriptions for maximizing behavior.

To fill this need, a programming method must be feasible and efficient on modern computers, and it must also be efficient in terms of human resources. Because man hours are as important as machine time, the method should be widely applicable with little need for modification, easily understood, and easily used.
Experience with the method will finally determine the value of the results reported here. Initial evidence indicates that it is feasible and efficient, and later discussions will show the method to be quite flexible. Although not trivial, it seems to be about as simple conceptually as one can expect. Later discussions provide comparisons with other methods, but technical difficulties prevent a firm conclusion about relative merits.

1.1 Results Obtained

The main result is a computational procedure for solving concave programming problems which satisfy certain differentiability conditions. Developed in a general form in Sections 2 and 4, this procedure is specialized to solve three important types of concave programming problems in Sections 3, 5, and 6. Tangible evidence that the procedure works is a set of three computer routines for solving these types of problems. With little modification in the computer routines one can solve virtually any differentiable concave programming problem.

Two theoretical developments make possible the construction of the computational procedure. The first development is a "Simplicial Algorithm" for quadratic programming, obtained as a special case of a generalization to concave programming of the Simplex Method used in linear programming. The general Simplicial Algorithm is described in Section 2, and then in Section 3 it is specialized to quadratic programming. The quadratic programming algorithm introduces the principle of the floating pivot, which besides making possible a simple tableau representation of the problem, contributes greatly to an efficient algorithm. In more general problems than quadratic programming
the Simplicial Algorithm encounters burdensome computations, but this difficulty is averted by the second development.

The second theoretical development extends the power of the Simplicial Algorithm from quadratic programming into a wider domain of problems. Newton's method of solving simultaneous equations is combined with the simplicial procedures to construct an algorithm which is feasible, and apparently efficient, for general concave programming problems satisfying certain differentiability conditions. No radical departure is involved here; yet, the combination of these two techniques is surprisingly effective. The basic principle at work is that quadratic approximations to the objective function and the constraint functions of the programming problem permit (approximate) determination of the optimal "basis" (defined in Section 2) by the simplicial procedures. Once the basis is known, Newton's method converges to a solution to whatever degree of approximation is desired.

Section 4 develops the programming method in the general case, and then Sections 5 and 6 describe its application in two special types of problems.

1.2 The Computer Routines

Among the concrete accomplishments of the research are computer routines for solving three important types of programming problems. The three problems are the following:

1. Quadratic Programming;
2. Linear Programming under Uncertainty;
3. Variable-Factor (or Blend) Programming.
The formulations of these problems are elaborated in Sections 3, 5, and 6, respectively.

Besides their individual significance, these problems were chosen as applications because they exhibit the programming method from different vantage points. Thus, the Simplicial Algorithm is represented in the quadratic programming routine without any of the complications of simultaneous nonlinear equations. The routine for linear programming under uncertainty exhibits the method for the special case of a nonlinear objective and linear constraints. The complementary situation with a linear objective and nonlinear constraints is demonstrated in the blending routine. (A problem with both a nonlinear objective and nonlinear constraints presents no difficulties beyond those encountered in the three types of problems treated.)

Each of the computer routines has been thoroughly tested. First tested on small problems for which the answers could feasibly be checked in detail by hand, each routine was then used to solve larger problems which were designed to be representative of realistic situations. Finally, each routine was applied to solve a host of problems for which the data was provided by a random number generator. Further details of these tests, examples of problems solved, and data on the computational characteristics of the routines, are included in later discussions.

Anyone with these types of problems to solve can use the computer routines. Complete instructions are included here, and copies of these instructions, plus copies of the program card decks, are deposited with Professor John Bishop. The object-program card
decks are to be used on an IBM 709 or 7090 machine; but recompilations for the FORTRAN source-program card decks suffice to create object-programs for any of a wide variety of machines, ranging from the IBM 1401 to the CDC 1604. In most cases, use of the computer routines requires nothing more than preparing the input data on punch-cards in appropriate fashion.

The practical potential of the research is evidenced in the computer routines. Besides showing that the method works, they are the first members of a programming package. By a programming package we mean a set of computer routines for solving the more important practical programming problems. Each of the three routines that I have developed is built around several core subroutines which accomplish the basic optimizing procedures. With these core subroutines already available, the development of a computer routine for any other differentiable, concave programming problem requires merely the writing of additional subroutines for (1.) data input, (2.) calculations of values of the functions involved, and (3.) output of the solution. Subroutines for these purposes are, of course, unique to each type of problem. Now that the basic theory and the core subroutines have been developed, further development of a rather complete programming package should be relatively easy and economical.

1.3 Relation to Other Work

The ideas in Kuhn and Tucker's original paper lead naturally to the general simplicial procedure developed here. Although my theoretical justification differs somewhat from Beale's justification, the simplicial algorithm in the special case of quadratic programming
with linear constraints is closely akin to his method for that problem. In this context the floating pivot technique can be construed as an extension of Beale's method. A comparison is made in Section 3.

Surveys of the several other general programming methods that have been proposed are given by Wolfe (1962) and by Dorn. The most prominent methods are the gradient projection method of Rosen, the cutting plane method of Kelley, and the method of feasible directions of Zoutendijk. Computer routines employing these methods are not available for direct comparisons, making it difficult to evaluate them in relation to the method proposed here.

No other computer routines are known for solving the particular problems treated in Sections 5 and 6.
2. The Simplicial Algorithm

Dantzig's Simplex Algorithm (1949) for linear programming is generalized in this section to solve any concave programming problem. By a simplicial algorithm, a terminology originating with Wolfe (1962), I mean a programming algorithm which uses the same method of solution as the Simplex Algorithm. The method can be characterized in this way: the search for a solution among all the possible candidates is confined to those candidates which are feasible, and which satisfy the complementary slackness condition (elaborated below). In the nomenclature of linear programming one would say that the search is confined to basic candidates, and this will be true here, after the concept of a basis is extended appropriately. An important feature of simplicial algorithms is that they find a solution after examining only a limited number of candidates. In addition, the Simplex Algorithm, as well as the general Simplicial Algorithm to be described here, is easy to understand, and to use, and possesses a strong rationale in programming theory.

The fundamental theorems of concave programming are presented without proofs in this section. From the theorems I construct a general Simplicial Algorithm. The algorithm uses only the intuitive ideas inherent in these theorems.

In Section 3 I will apply the Simplicial Algorithm to the special case of quadratic programming, for which a substantial refinement of the computational technique is possible. Newton's
method is introduced in Section 4 to construct a variant of the Simplicial Algorithm which is more efficient for nonquadratic programming problems.

2.1 Formulation of the Concave Programming Problem, and Notational Conventions

Our subject matter, as formulated originally by H. W. Kuhn and A. W. Tucker, is the

Concave Programming Problem: Find a value of the program vector \( x = (x_j) \), in the \( J \)-dimensional real domain, yielding the maximal value of the objective function \( g(x) \) subject to the nonnegativity constraints \( x_j \geq 0 \) for \( j = 1, \ldots, J \), on the program vector and the \( N \) additional constraints \( f_n(x) \geq 0 \), for \( n = 1, \ldots, N \), where \( g \) and each \( f_n \) is real-valued and concave.

For our purposes it suffices that \( g \) and \( f_n \) be defined only over the feasible subspace:

\[
F = \{ x | x \geq 0, \quad f_n(x) \geq 0, \quad n = 1, \ldots, N \} 
\]  \hspace{1cm} (2-1)

A function such as \( g \), or any \( f_n \), is concave if linear interpolation between the values at any two points of its range yields a value not greater than its actual value at the point of interpolation; viz.,

\[
ag(x^1) + (1-a) g(x^2) \leq g(ax^1 + (1-a)x^2), \quad 0 \leq a \leq 1, \hspace{1cm} (2-2)
\]

for every \( x^1 \) and \( x^2 \) in the domain. Among the many references on
concavity, and the related concept of convexity, one can consult
Fenchel, Eggleson, or Appendix B of Karlin, where the properties of
concave functions can be found.

A concave function possesses partial derivatives almost
everywhere, so we can use the gradient operator \( \nabla \) defined by

\[
\nabla g(x) = \left( \frac{\partial g(x)}{\partial x_1}, \ldots, \frac{\partial g(x)}{\partial x_j} \right)^t
\]

(2-3)

to transform the scalar valued function \( g \) into the vector valued
function \( \nabla g \) of the same argument. At a point where a concave
function does not have some partial derivative the left and right
partial derivatives will be defined, and we will assume that at the
time required some proportional combination of left and right partial
derivatives will be specified appropriate to the circumstances. It
should be noted that gradient vectors are column vectors, as are all
the other vectors used in this paper.

Extending the notation further, we define the gradient
matrix of a vector valued function, say \( f(x) = (f_n(x)) \), as the
\( J \) by \( N \) matrix

\[
\nabla f(x) = (\nabla f_n(x)) = \left( \frac{\partial f_n(x)}{\partial x_j} \right)
\]

(2-4)

where \( n = 1, \ldots, N \) indexes the component gradient vectors across
the columns.

Using the gradient notation, a scalar or vector valued
function \( f \) is concave if and only if

\[
f(x^2) \leq f(x^1) + \nabla f(x^1)^t[ x^2 - x^1 ]
\]

(2-5)
for all $x^1$ and $x^2$. If $f$ in (2-5) is interpreted as a production function, then this relation expresses the proposition that, in economic terms, a concave function exhibits nonincreasing marginal productivity.

2.2 Conditions for a Solution

H. W. Kuhn and A. W. Tucker proved the fundamental theorems of concave programming 1950. Although much work has been done since then, it is not necessary here to recount the later developments. A compendium of work in the area is given by Vajda, while Karlin displays the elegance and rigor of the present theory. The relevant theorems, all due to Kuhn and Tucker, are stated here without proofs. To these theorems I will add an important lemma in a later discussion.

The substantive proposition of the Kuhn-Tucker theory demonstrates an equivalence between the concave programming problem and a related saddle problem.

**Definition:** The saddle problem with objective function $G(y; u)$ is to find vectors $x^0 \geq 0$ and $u^0 \geq 0$ for which

$$G(x; u^0) \leq G(x^0; u^0) \leq G(x^0; u)$$

for all $x \geq 0$ and $u \geq 0$.

For the economic interpretations of the saddle problem see, for example, the comments of Kuhn and Tucker, or in limited contexts Samuelson, Dorfman, and Dorfman, Samuelson, and Solow.

The main accomplishment of the theory is to identify a form of the objective function in the saddle problem such that the programming problem and the saddle problem are equivalent. From this equivalence one can then deduce conditions for a solution. That any solution $(x^0; u^0)$ to the saddle problem yields $x^0$ as a solution to the programming problem
is immediate, even without the concavity assumptions. The more difficult task is an existence theorem showing that under certain conditions there will exist a solution \((x^0; u^0)\) to the saddle problem for every solution \(x^0\) to the programming problem. The concavity assumption is needed for this theorem; in addition, it is necessary to impose a regularity condition on the constraints in the form of a

Constraint Qualification: Let \(H(x)\) be the matrix formed by appending the \(J\)-dimensional identity matrix to \(-\nabla f(x)\), as

\[
H(x) = [-\nabla f(x), I],
\]

and define \(\hat{H}(x)\) to be any \(J\) by \(J\) matrix obtained from \(H(x)\) by deleting any \(M\) columns; then it is assumed that every \(\hat{H}(x)\) is non-singular for all \(x\) in the feasible subspace \(F\).

The role of the constraint qualification is demonstrated by Kuhn and Tucker who formulate it in different terms, and by Wilde. Always satisfied by independent linear constraints, the constraint qualification will also be satisfied whenever there exists a program vector for which no constraints are binding; viz., there exists \(\bar{x} >_0 0\) for which \(f(\bar{x}) > 0\). The constraint qualification is only mildly restrictive, and is assumed to be satisfied in all the work in this paper.

The following are the fundamental theorems of concave programming proved by Kuhn and Tucker.

Equivalence Theorem: \(x^0\) is a solution to the concave programming problem if and only if there exists a solution \((x^0; u^0)\) to the saddle problem with objective function

\[
G(x; u) = g(x) + u^t f(x).
\]
This theorem establishes the desired equivalence relation, so that an investigation of the programming problem can be conducted via the saddle problem. Due to the concavity assumptions, the differential conditions for a solution to the saddle problem are sufficient as well as necessary, as stated in the next

Theorem: \((x^0; u^0)\) is a solution to the saddle problem of concave programming if and only if

\[
\begin{align*}
\text{a)} & \quad x^0 \geq 0, \quad \nabla g(x^0) + \nabla f(x^0)u^0 \leq 0, \quad [\nabla g(x^0) + \nabla f(x^0)u^0]^t x^0 = 0; \\
\text{b)} & \quad u^0 \geq 0, \quad f(x^0) \geq 0, \quad u^0^t f(x^0) = 0.
\end{align*}
\]

This theorem states a mutual exclusiveness between a variable and the corresponding slack in the inequalities of the differential conditions for a solution. Such a relation between variables is formalized in the

Complementary Slackness Condition: Two variables \(x_j\) and \(v_j\) satisfy the complementary slackness condition when one of them is constrained identically to zero.

Two vectors \(x\) and \(v\) satisfy the complementary slackness condition when each component pair \(x_j\) and \(v_j\) satisfies it. Introducing slack vectors \(v\) and \(s\) into the right inequalities of (a) and (b) above, respectively, the previous results combine to yield the

Major Theorem: \(x^0\) is a solution to the concave programming problem if and only if there exist \(M\)-dimensional vectors \(s^0\) and \(u^0\), and a \(J\)-dimensional vector \(v^0\), such that

\[
\begin{align*}
(1) & \quad x^0, s^0, v^0, \text{ and } u^0 \text{ are all nonnegative;} \\
(2) & \quad \begin{bmatrix} x^0 \\ s^0 \end{bmatrix} \text{ and } \begin{bmatrix} v^0 \\ u^0 \end{bmatrix} \text{ satisfy the complementary slackness condition;} \\
(3) & \quad x^0, s^0, v^0, u^0 \text{ provide a solution to the following set of } J + M \text{ equations in } 2(J + M) \text{ variables,}
\end{align*}
\]

\[ \nabla g(x) + v + \nabla f(x)u = 0 \]
\[ -f(x) + a = 0 \] (2-6)

The operational aspects of the Kuhn-Tucker results are summarized in this Major Theorem. To put it briefly, it states that the computational task is to find a solution to the simultaneous equations (2-6), which satisfies as well (i) the nonnegativity condition and (ii) the complementary slackness condition.

Were it not for the nonnegativity condition (i) and its derivative, the complementary slackness condition (ii), the Kuhn-Tucker results would be the means as well as the end of concave programming. As it is, however, the conditions for a solution cannot be resolved directly, due to combinatorial complexities. Under the circumstances, a systematic search procedure is needed.

2.3. Nature of the Computational Task

Two techniques are needed for solving a concave programming problem: one is a way of solving the simultaneous equations (2-6), and the other is a procedure for searching for a solution to the programming problem among the solutions to (2-6). My immediate concern in this section is the search procedure. In order to concentrate on the subject at hand, assume that the algebraic techniques required are all accomplished by a "black box". I will specify what goes on inside this black box in Sections 3 and 4.

The nature of the computational task is implicit in various ways in the Major Theorem, at least three approaches being possible. One way is to ignore, for the most part, the conditions for a solution except as a check for optimality at the end. Most of the algorithms
(cf. Wolfe (1962)) based on gradient procedures do this in effect, rely-
ing on feasible directions of steepest ascent as their path of search, and as the means of solution.

A second way is to identify all solutions to (2-6) and then to search among them for an optimal candidate; unfortunately, very little work has been done in this direction.

The third way, the approach of the Simplicial Algorithm, is to satisfy part of the conditions for a solution, and then systematically to search for an optimal candidate by making adjustments in the remaining conditions until they are all satisfied. In practice, the conditions (ii) and (iii) of the Major Theorem are easy to satisfy initially, and it is usually relatively easy to satisfy part of condition (i) also. I shall discuss the Simplicial Algorithm for only one way of satisfying part of the nonnegativity condition (i); viz., when an initial candidate is available for which \( x \geq 0, s \geq 0 \).

2.4. The Notion of a Basis, and Some Terminology

In order to obtain a solution to (2-6) which satisfies the comple-
mentary slackness condition (ii) of the Major Theorem, one needs merely to specify for each pair of complementary variables \( x_j \) and \( v_j \), and \( s_n \) and \( u_n \) which member of the pair is to be constrained to zero. There must then be \( J + N \) unconstrained variables in terms of which the \( J + N \) simultaneous equation (2-6) can be solved.

The unconstrained variables will be called basic variables, and the others, nonbasic variables. A basis is then defined as a set of indices of the vector \((x^t, s^t, v^t, u^t)^t\). When it is unqualified, the
term basis presumes that the complementary slackness condition is satisfied, so that if the index \( i \leq J + N \) is in the basis then \( i + J + N \) is not in the basis, and vice versa. For a semibasis the complementary slackness condition need not be satisfied, but otherwise the definition is the same.

Given some semibasis, the values of the basic variables will be denoted by a superimposed carat. Then a basic candidate is a quadruplet \((\hat{z}; \hat{v}; \hat{\theta}; \hat{u})\) of vectors of values of basic variables. Being constrained to zero, the nonbasic variables are omitted from the formulation.

The values of the basic variables are determined by solution of (2-6). The Major Theorem assures that a solution to these equations will exist for the optimal choice of a basis, provided a finite solution exists to the programming problem. No such assurance is given for a non-optimal basis, but this presents no difficulties in practice since the search procedure moves from any one basis consistent with (2-6) to some other which, by construction, is also consistent.

We shall call the \( x \) and \( s \) variables primal variables, and the \( u \) and \( v \) variables dual variables. The technical terminology calls \( x \) and \( u \) the primal and dual program variables, respectively; and \( s \) and \( v \), the primal and dual slack variables, respectively. Because the differentiation between program and slack variables is not always necessary, we shall often use \( y = (x^t, s^t)^t \) and \( v = (v^t, u^t)^t \) to denote the vectors of primal and dual variables, respectively.

A feasible candidate will be taken to be a primal-feasible candidate; i.e., a candidate \((y; w)\) for which \( y \geq 0 \).
2.5. The Significance of Dual Variables; A Lemma

More information is needed in order to make the Kuhn-Tucker results operational. The Major Theorem provides a test for accepting or rejecting candidates for a solution, but it does not indicate a criterion for improving one’s choice. In particular, one needs to know which nonbasic variables can beneficially be made basic and/or which basic variables might be made nonbasic.

Partially formulated by Dreyfus and Freimer and proved by Wilde, a simple lemma yields the required information.

**Lemma:** Given a basis for which $y_j$ is a nonbasic primal variable,

$$\frac{\partial g(\bar{y})}{\partial y_j} = -\bar{\alpha}_j .$$

The hypothesis requires that the complementary slackness condition be satisfied by $(\bar{y}; \bar{u})$ in the sense that (1) the nonbasic variables are all specified as constants, or parameters, in (2-5), and (2) of two complementary variables, $y_j$ and $v_j$, one must be basic and the other nonbasic. The values of the basic variables are assumed to be determined by (2-6). The result says that as the nonbasic primal variable $y_j$ is increased, all the while keeping the other nonbasic variables constrained to zero, and keeping the basic variables so as to satisfy (2-6) given $y_j$ as a parameter of the system, that the rate of change in the objective function is the negative of the complementary basic dual variable.

Specifically, if the constraint $l$ were to be tightened by a small increment $\Delta y_j$, then to a first approximation the change in the value of the objective function would be $-\bar{\alpha}_j \Delta y_j$. Hence, it pays to increase a nonbasic primal variable for which the corresponding basic dual variable is negative.
2.6. The General Simplicial Procedure

Let me outline the algorithm in terms of the ideas that have already been discussed. We will start from an initial basis for which solution of (2-6) yields a feasible candidate, \( \hat{y} \geq 0 \). (Methods of obtaining an initial feasible basis are described by Wolfe (1962).) If the candidate is also dual feasible, \( \hat{w} > 0 \), then the basis is optimal, according to the Major Theorem. Otherwise, we will make basis changes one at a time to eliminate dual infeasibility, while retaining primal feasibility, until an optimal basis is obtained. The key to the algorithm is the lemma, since it indicates how to change the basis so as to improve the attained value of the objective function.

The procedure can be outlined roughly as follows: Suppose \((\hat{y}, \hat{w})\) is a basic feasible candidate, \( \hat{y} \geq 0 \) but \( \hat{w} \) is not nonnegative. In this case there is no loss of generality if we choose the labeling so that \( \hat{w}_1 < 0 \). According to the lemma, increasing the nonbasic primal variable \( y_1 \) will increase the objective function. Hence we proceed by releasing the constraint \( y_1 = 0 \) and drive, \( \hat{w}_1 \), so to speak, towards zero assuming that \((\hat{y}, \hat{w})\) is kept in balance to satisfy (2-6), and that all nonbasic variables other than \( y_1 \) remain at zero. As we push \( \hat{w}_1 \) towards zero \( \hat{y} \) changes continuously. If we can push \( \hat{w}_1 \) all the way to zero keeping \( \hat{y} \geq 0 \) then we arrive at a new basis where \( \hat{y}_1 > 0 \), and \( \hat{w}_1 = 0 \). If, however, we cannot push \( \hat{w}_1 \) all the way towards zero maintaining feasibility of the \( \hat{y} \) then we can assume that the labeling is chosen so that \( \hat{y}_2 \) is the first basic primal variable to hit zero. We can now continue driving \( \hat{w}_1 \) towards zero by holding fast on \( y_2 = 0 \) but now releasing \( w_2 \). In
this way we can continue until all the basic dual variables, as well as the basic primal variables, are nonnegative and a solution has been found. Now I shall elaborate the details.

Select a basic dual variable which is negative, say $\hat{\xi}_1 < 0$. According to the lemma, increasing the complementary nonbasic primal variable, $y_1$, will increase the objective function, so we want to increase $y_1$ as long as it is beneficial and feasible. Our task, then, is to ascertain two critical values of $y_1$; viz.,

1. $\check{y}_1$, the critical value at which it is no longer beneficial to increase $y_1$; and
2. $\bar{y}_1$, the critical value at which it is no longer feasible to increase $y_1$.

The smaller of these two critical values will dictate the change of basis to be made. The first critical value is easily determined from the lemma, but the second critical value introduces some complications.

The first critical value will be that value of $y_1$ for which a further increase would not increase the objective function. Consequently, applying the lemma again it is that value for which $\hat{\xi}_1$ becomes zero. If no positive value of $y_1$ will drive $\hat{\xi}_1$ to zero, then $\bar{y}_1$ is unbounded. When $\check{y}_1$ is less than $\bar{y}_1$ it is clear that we will make $v_1$ nonbasic and replace it by making $y_1$ basic: unless this yields an optimal basis, the entire procedure will then be repeated.

To find $\bar{y}_1$, we increase $y_1$ in the same manner as above until some basic primal variable, say $\hat{y}_2$, passes zero to become negative. That is, we want to increase $y_1$ as far as it is feasible to do so. It will no longer be feasible to increase $y_1$ further when some basic primal variable
becomes negative. By identifying $\bar{y}_1$ as the smallest positive value of $y_1$ for which some basic primal variable, $\bar{y}_2$, becomes zero, we are specifying the maximum amount that $y_1$ can be increased. This much is clear; what is not so clear is how the basis is to be reconstituted. The situation is such that we will find it desirable to proceed, not directly from basis to basis, but from basis to basis with intermediate stops at semibases.

The reasoning is as follows. If $\bar{y}_1$ is less than $\bar{y}_1$, then we know that $\bar{y}_1$ is still negative at $y_1 = \bar{y}_1$, and the attained value of the objective function can still be improved by increasing $y_1$ beyond $\bar{y}_1$. To increase $y_1$ further, however, would necessitate appending the constraint $\bar{y}_2 = 0$ to (2-6) in order to avoid becoming infeasible, complicating the technique drastically. A simple trick, however, alleviates the difficulty. Specify the constraint on $y_2$ by making it a nonbasic variable and replace it with $y_1$ as a basic variable. Although this yields a semibasis, it is not a proper basis because $y_1$ and $y_1$ are both basic variables, and $y_2$ and $y_2$ are both nonbasic variables. But since $y_2$ is nonbasic we are now free to increase $y_2$. This allows us to increase $y_1$ and $y_1$ further, not by increasing $y_1$ directly since it is now basic and is a dependent variable determined by (2-6), but by increasing $y_2$ as the controllable parameter in the system. (The fact that we want to increase $y_2$, rather than to decrease it, in order to drive $\bar{y}_1$ to zero is seen by the fact that if $\bar{y}_1$ were forced to zero by making $y_1$ nonbasic and $y_2$ basic, then the attained value of $y_2$ would be $\bar{y}_2 = - \partial g / \partial y_2$, which must be positive since we have just previously found that a beneficial increase in $y_1$ was prevented by $\bar{y}_1$. \end{document}
becoming negative.) Hence, we must do for \( w_2 \) what we did for \( y_1 \); viz.,
calculate the two critical values \( \bar{v}_2 \) (at which \( \hat{v}_1 \) is driven to zero)
and \( \bar{v}_2 \) (at which some basic primal variable, say \( \hat{y}_3 \), goes to zero) and
repeat the analysis. In this way we can go from a basis through a
sequence of semibases until finally \( \hat{v}_1 \) can feasibly be driven to zero.
At that stage there is some nonbasic dual variable, say \( \hat{w}_4 \), which is the
parameter in (2-6), and we can reconstitute the basis by making \( \hat{w}_4 \) basic
and \( \hat{v}_1 \) nonbasic. From there the entire procedure can be repeated until
an optimal basis is found.

If at any stage either critical value is negative or is inconsistent
with (2-6), then it is taken to be infinite. If a critical value is zero
then the problem is possibly degenerate (in the same sense as in linear pro-
gramming (Gass)) and it is well to perturbate the data of the problem
slightly to remove the indeterminacy.

This is the substance of the simplicial search procedure, although
a good deal remains to be said about the algebraic technique required to
implement it. Sections 3-6 are concerned mainly with the algebraic tech-
nique. Omitting the details, we can formalize the search procedure in the
flow chart shown in Figure 2-1.
Figure 2-1

1. Start by choosing a feasible initial basis.
2. Calculate values of basic variables from (2-6).
3. Are basic variables all nonnegative? Yes → Stop
   No →
4. Choose a negative basic dual variable, and identify its complementary (nonbasic primal) variable as the parametric variable.
5. Increase the parametric variable in (2-6) to find its two critical values (Stop if both are unbounded).
6. Is the first critical value the smaller? Yes → Step 8
   No →
7. Make the parametric variable a basic variable to replace the basic primal variable which became zero, identify the latter's complementary (nonbasic dual) variable as the new parametric variable and return to 5.
8. Make the parametric variable a basic variable to replace the negative basic dual variable, and return to 2.

2.7. A Geometrical Description

A geometrical description of the search procedure highlights some of its important features.

Figure 2-2 depicts a representative feasible subspace for J = 2, N = 3. The boundaries of the various constraints are represented by the curves for which \( x_1 = 0, x_2 = 0, s_1 = 0, s_2 = 0, \) and \( s_3 = 0, \) respectively, given that the primal variables are related by the system,

\[
-\mathbf{f}_1(x) + s_1 = 0, \\
-\mathbf{f}_2(x) + s_2 = 0, \\
-\mathbf{f}_3(x) + s_3 = 0.
\]

The feasible subspace is the set of primal program vectors (points in Figure 2-2) for which all the primal variables are nonnegative. The objective function attains an absolute maximum at \( P_4, \) and at \( P_3 \) when constrained only by \( f_1(x) = 0. \) The dotted curves in Figure 2-2 are isoquants of the objective function. The segmented curve \( P_0P_1P_2P_4 \) will be the path to the solution from the initial candidate \( P_0. \)
The Simplicial Algorithm operates as follows. For the initial basis,

\[ B_0 = (3, 4, 5, 6, 7) \],

the slack variables are all basic, and the initial candidate is the origin, \( P_0 \), in the space of primal program variables. This basis is feasible because \( \hat{s}_1 > 0 \), \( \hat{s}_2 > 0 \), and \( \hat{s}_3 > 0 \). It is not optimal, however, since \( \hat{v}_2 < 0 \). Letting \( x_2 \) be a parameter in the system (2-6), which here takes the form

\[
\begin{align*}
\nabla g(x_1 = 0, x_2) \quad + \hat{v} = 0, \\
-\lambda(x_1 = 0, x_2) + \hat{s} = 0,
\end{align*}
\]
yields $\bar{x}_2 < \bar{x}_2$. Hence, $v_2$ is made nonbasic and $x_2$ becomes basic to attain the point $P_1$ at the end of the first iteration.

The second iteration begins by finding $\hat{v}_1 < 0$. Now letting $x_1$ be a parameter in (2-6) allows one to traverse the line from $P_1$ to $P_2$. Since this line is defined by the condition $v_2 = 0$, or equivalently $\partial g(x_1, x_2)/\partial x_2 = 0$, the extension of this line necessarily passes through $Q$. Calculating the critical values of $x_1$ yields $\bar{x}_1$ as the value at which $Q$ would be attained, and $\bar{x}_1$ as the value at which $P_2$ would be attained. Since $\bar{x}_1 < \bar{x}_1$, and $\partial_1 = 0$ for $x_1 = \bar{x}_1$, $s_1$ is made nonbasic and $x_1$ becomes basic. This creates a semibasis for which $u_1$ is to be the parameter. Note that $\hat{v}_1$ must still be negative.

Continuing to drive $\hat{v}_1$ to zero, the third iteration seeks the critical values of $u_1$. Since $s_1$ is now nonbasic, varying the parameter $u_1$ must make us traverse the curve $s_1 = 0$. Several facts are now evident. First, as $u_1$ is varied $\hat{v}_1$ must become zero at $P_3$, since that is the point at which the maximum of the objective function occurs as $x_1$ varies subject to $s_1 = 0$. This identifies the value of $\bar{u}_1$. Second, it must always be that $\bar{u}_1 > 0$, since if the point $P_3$ were attained by making $v_1$ nonbasic and $u_1$ basic, then one would have $\partial g/\partial s_1 = -\bar{u}_1$ which would have to be negative. Increasing $u_1$ as a parameter in (2-6), then, we obtain $\bar{u}_1$ yielding $P_4$ for which $\bar{s}_2 = 0$, and $\bar{u}_1$ yielding $P_3$. Because $\bar{u}_1 < \bar{u}_1$ we make $s_2$ nonbasic and $u_1$ basic.

In this example we know that the final iteration will consist of making $u_2$ basic and $v_1$ nonbasic. This is because $P_4$ is a vertex of the feasible subspace, so that there are no degrees of freedom allowing the
primal variables to vary as \( u_2 \) is varied; hence \( \bar{u}_2 \) is unbounded and it must be that \( \bar{u}_2 < \bar{u}_2^* \).

A small numerical example illustrating the type of algebraic manipulations required by the Simplicial Algorithm is solved in Appendix A.
3. Quadratic Programming

The Simplicial Algorithm for quadratic programming is described in this section. My main concern here are the details of the algebraic technique, since the general algorithm was described in Section 2. In this special case linear algebra suffices, so that a simple and efficient computational scheme is possible.

Among the reported problems to which quadratic programming has been applied are portfolio analysis (Markowitz), pricing and activity analysis when demand curves are linear (Dorfman), and allocation of a strategic material (Karrerman). Quadratic programming has a special role in this paper: in Section 4 I will present a method of solving more general concave programming problems, in effect, by solving a succession of approximating quadratic problems.

3.1. Introduction

The quadratic programming problem is a special case of the concave programming problem defined in Section 2, in which the objective is a quadratic function, and the constraints are linear functions. The notation we shall use is specified in the following statement of the

**Quadratic Programming Problem:** solve the concave programming problem when the objective function is

\[ g(x) = p^T x + x^T Ax , \]

and the constraint function is

\[ f(x) = c - Dx . \]

The quadratic form must be negative semidefinite over the feasible subspace in order to satisfy the concavity assumptions.
Recall from the formulation of the Simplicial Algorithm in Section 2 (summarized in the flow chart in Figure 2-1), that there are two algebraic operations required by the algorithm. In each iteration one must do the following:

1. Given a nonbasic parametric variable which is to be increased, determine the two critical values at which increasing the parametric variable further is no longer beneficial, and feasible, respectively;

2. Given the basic variable which is to be made nonbasic as determined by the smaller of the two critical values, make the parametric variable basic and determine the new values of all the basic variables. The first operation is called the search procedure, and the second is called the pivoting operation. In the search procedure we are searching for the basic variable to be made nonbasic; the name of the pivoting operation comes to us from linear algebra. Both operations are familiar from the Simplex Algorithm for linear programming. They are described below in sequence after a preliminary discussion in which the computational task is cast in terms of linear algebra, and some appropriate notation is established.

The discussions below will, in effect, be specializing the flow chart of the general algorithm in Figure 2-1 to the complete flow chart for quadratic programming in Figure 3-1. The reader might find an occasional look at these flow charts helpful in recalling how the pieces discussed below fit together.

3.2. The Basic Representation

The algebraic operations of the Simplicial Algorithm are mostly concerned with finding and analyzing basic candidates determined as solutions
to the simultaneous equations (2-6) in the Major Theorem. For the special case of quadratic programming these equations are linear, taking the following form:

\[
\begin{bmatrix}
A + A^t & 0 & I_j & -D^t \\
D & I_N & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
v \\
u \\
c
\end{bmatrix}
= \begin{bmatrix}
-p \\
0
\end{bmatrix}.
\] (3-1)

If we consider the matrix on the left as a collection of column vectors, then we can regard (3-1) as requiring us to find a representation of the vector on the right as a linear combination of the vectors in the matrix, or just of the basic vectors when a semibasis is given for the programming problem. Thus, the basic vectors form a basis for a vector space over which the representation is to be obtained, where the term basis is understood to indicate that it is interpreted in the sense of linear algebra (cf. Ordin).

To see where these ideas lead us, let us adjoin the vector on the right to the matrix in (3-1) and call the resulting \((J + N)\) by 2\((J + N) + 1\) matrix \(R = (r_{ij})\); viz.,

\[
R = \begin{bmatrix}
A + A^t & 0 & I_j & -D^t & -p \\
D & I_N & 0 & 0 & c
\end{bmatrix}.
\] (3-2)

For notational convenience we will reorder the columns of \(R\) so that the basic vectors occur on the left, as

\[
R = \begin{bmatrix}
R_B & R_N & r
\end{bmatrix},
\] (3-3)

where \(R_B\) is the matrix of basic vectors in some order, \(R_N\) is the matrix of nonbasic vectors in the same order, and \(r\) is the last column of (3-2). Now \(R\) can be represented formally as a collection of column vectors represented over the unit vectors,
\[ R = IR, \quad (3-1) \]

where \( I \) is the \((J + N)\)-dimensional identity matrix. We can see, however, that what we want to work with is the representation, say \( \hat{R} \), of \( R \) over the basic vectors \( R_B \),

\[ R = R_B \hat{R}; \quad (3-5) \]

for,

\[ \hat{R} = \begin{bmatrix} I, & R_B^{-1} R_N, & R_B^{-1} r \end{bmatrix}, \quad (3-6) \]

which yields us the values of the basic variables in the last column.

Given any semibasis for the programming problem, we shall call the representation of \( R \) over the basic vectors the basic representation, and it will be denoted by \( \hat{R} = (\hat{r}_{ij}) \). Note that \( \hat{R} \) is the basic representation relative to whatever semibasis is currently at hand in the course of the algorithm.

3.5. The Search Procedure

Let us first recall the role of the critical values. As we begin the search for the basic variable to be made nonbasic, we will have on hand a negative basic dual variable which we are driving to zero. This basic dual variable we will call the floating pivot variable, the reason being that possibly this variable will be with us for several iterations as we try repeatedly to drive it to zero, and then when finally it can feasibly be driven to zero it will be the variable made nonbasic when we execute the pivoting operation. We will also have found a nonbasic variable, called the parametric variable, which is to be increased as long as it is beneficial and feasible. The parametric variable will be a primal variable when a proper basis obtains, and otherwise it will be
a dual variable; hence, rather than use the \((y, w)\) notation of Section 2, we will here represent the parametric variable as \(z_p\), where \(p\) is the index of its column in \(\hat{\mathbf{R}}\). The two critical values of the parametric variable are \(\bar{z}_p\), at which no further increase is beneficial; and \(\bar{z}_p'\), at which no further increase is feasible, as determined by the fact that some basic primal variable becomes negative.

In order to determine the critical values, let \(\mathbf{z}\) and \(\bar{\mathbf{z}}\) be the vectors of values of basic and nonbasic variables, respectively, and observe from (3-1) that

\[
\mathbf{z} = \mathbf{R}^{-1} \mathbf{r} - \mathbf{R}^{-1} \mathbf{R} \mathbf{e} \bar{\mathbf{z}}.
\]

(3-7)

Now as \(\bar{z}_p\) is increased,

\[
\mathbf{z} = \mathbf{R}_L - \mathbf{R}_P \mathbf{z}_p,
\]

(3-8)

where from (3-6), \(\mathbf{R}_L\) and \(\mathbf{R}_P\) are the appropriate columns of \(\mathbf{R}\), and we let the index \(L = 2(J + N) + 1\) signify the last column. The formulas for the critical values now follow immediately from (3-8).

1. Let \(f\) be the index of the row in \(\hat{\mathbf{R}}\) of the floating pivot variable. Then \(\bar{z}_p\), the value of \(z_p\) at which the floating pivot variable becomes zero, is

\[
\bar{z}_p = \bar{p}_f / \bar{p}_p
\]

(3-9)

provided \(\bar{p}_p < 0\), and it is unbounded otherwise (\(\bar{p}_f\), the current value of the floating pivot variable is negative by construction).

2. The first value of \(z_p\) at which a further increase is infeasible is \(\bar{z}_p'\), determined as the minimum nonnegative ratio \(\bar{p}_i / \bar{p}_p'\) where \(i\) ranges over the rows of those basic primal variables for which \(\bar{p}_p > 0\).
These easy formulas for the critical values direct one to go down
column p of \(\bar{B}\), and for each row \(i\) for which either 1) \(l = f\) and \(\bar{P}_{ip} < 0\),
or 2) \(i\) indexes the row of a basic primal variable and \(\bar{P}_{ip} > 0\), calculate
the ratio \(\bar{P}_{iL}/\bar{P}_{ip}\). The minimum nonnegative ratio then identifies the
basic variable to be made nonbasic. If no such ratio exists, then the
solution is unbounded with the parametric variable increased to infinity.
If the minimum is zero, then the problem is degenerate and it is well to
perturbate that \(\bar{P}_{iL} = 0\) to a small positive value \(\bar{P}_{iL} = \varepsilon\) before continuing.

3.4. The Pivoting Operation

What we need for the pivoting operation is a formula that will take
the row index, say it is \(b\), of the basic variable to be made nonbasic,
and the column index \(p\) of the parametric variable to be made basic, as
input data and then transform the current basic representation into the
new basic representation. This operation might be depicted schematically
as follows:

\[
\hat{\bar{R}} = P(\bar{b}, \bar{p}) \hat{\bar{R}}, \tag{3-10}
\]

where \(P\) denotes the pivoting operation. In theoretical terms the job
of the pivoting operation is to replace a basic vector by a nonbasic
vector in the basis over which \(\hat{\bar{R}}\) is represented. The arithmetic details
of such a formula are available to us from linear algebra (cf. Orden),
and they have been applied previously to linear programming by Dantzig,
and by Dantzig, Orden and Wolfe.

Pivoting Operation: For a pivot column \(p\) defined by a nonbasic
vector, and a pivot row \(b\) defined by a basic vector, transform
the matrix \(\hat{\bar{R}} = (\hat{\bar{R}}_{1j})\) as follows, in the indicated order:
\[(1) \quad \tilde{p}_{ip} \leftarrow \tilde{p}_{ip} / \tilde{p}_{bp}, \quad 1 \neq b; \]
\[(11) \quad \tilde{p}_{ij} \leftarrow \tilde{p}_{ij} + \tilde{p}_{ip} \tilde{p}_{bj}, \quad 1 \neq b, \quad j \neq p; \]
\[(iii) \quad \tilde{p}_{bj} \leftarrow \tilde{p}_{bj} / \tilde{p}_{bp}, \quad j \neq p; \]
\[(iv) \quad \tilde{p}_{bp} \leftarrow 1/\tilde{p}_{bp}. \]

The operation is, of course, not defined when the pivot element \(p_{bp}\) is zero.

It might be noted here that the formulas for the critical values determined earlier are a reflection of Step (iii) in the pivoting operation, since the critical values are actually just the values the parametric variable would assume for various possible basis changes.

3.5. The Initial Basic Representation

The reader is now in a position to understand all of the flow chart in Figure 3-1 except Step 1, which will be explained below.

The first step of the algorithm is to establish the basic representation for the initial basis. This can be done using the pivoting operation by obtaining a sequence of relations equivalent to the transition from (3-4) to (3-5). Using the pivoting operation, one replaces the unit vectors in the basis of \(R\) one after another by the basic vectors in \(R_B\) to obtain the sequence.

\[
R = IR = B^0 R^0 \\
R = B^1 R^1 \\
\vdots \\
R = B^{J+W} R^{J+W} = R_B R, \tag{3-11}
\]

where

(a) \(B^0\) is the matrix of vectors in the basis after \(p\) unit vectors have been replaced, with \(B^0 = I\) and \(B^{J+W} = R_B\).
and

(b) $R^p$ is the representation of $R$ over the basis defined by $B^p$, with $R^o \cong R$ and $R^o \cup M \cong \tilde{R}$.

The computational scheme yielding this sequence is defined by the

Initialization Procedure: Perform the pivoting operation on $R^{p-1}$ to obtain $R^p$ for $p = 1, 2, \ldots, (J+N)$, where for each $p$
the pivot row $b$ can be any row not previously chosen for which
the pivot element is nonzero.

A sequence of nonzero pivot elements must exist if $R_B$ is nonsingular
(Orden), which we impose as a condition on the choice of the initial
basis. If $b \neq p$ then, as shown by Orden, the rows are permuted so that
row $b$ is relabeled as row $p$ (this is done often in practice, since choos-
ing the pivot element which is largest in absolute value minimizes round-
off errors).
Figure 3-1

Flow Chart of the Simplicial Algorithm for Quadratic Programming

0. Obtain an initial feasible basis.

1. To obtain \( \hat{R} \), establish the initial basis via the sequence (3-7)
   by executing the pivoting operation* on \( R^{p-1} \) for \( p = 1, \ldots, (J + N) \),
   for each \( p \) letting \( b \) be the row among those not previously used
   for which \( r_{bp}^{p-1} \) is largest in absolute value; if \( b \neq p \), then relabel
   row \( b \) as row \( p \).

2. Identify the most negative† value (of dual variables) in column
   \( L = 2(J + N) + 1 \) of \( \hat{R} \), designating that variable as the floating
   pivot variable, and its complementary nonbasic variable as the
   parametric variable; if no negative values are found, terminate
   the algorithm.

3. Let \( p \) be the column of the parametric variable, and find the mini-
   mum nonnegative ratio \( \hat{r}_{iL}/\hat{r}_{ip} \) for \( i \) ranging over the rows of the
   basic primal variables and the row of the floating pivot variable
   (if this minimum is zero for some \( i \) perturbate \( \hat{r}_{iL} \) by a small \( \epsilon > 0 \)
   and repeat; if no nonnegative ratios exist terminate with an unbounded
   solution); then designate the variable for which the minimum occurs
   as the basic pivot variable.

---

* Because the columns of the basic vectors are not needed, the pivoting
  operation for each \( p \) can be skipped for those \( j < p \).

† We select the most negative value so that the objective function will
  increase fastest, at least initially, as the variable is driven to zero.
4. Execute the pivoting operation* on $\hat{R}$ for $p$ the column of the parametric variable, and $b$ the row of the basic pivot variable; then relabel row $b$ as the row of the variable which is the parametric variable, and relabel column $p$ as the column of the variable which is the basic pivot variable.

5. If the floating pivot variable and the basic pivot variable are the same, then return to Step 2; otherwise, identify the variable complementary to the basic pivot variable as the new parametric variable and return to Step 3.

*The calculations can be skipped for those $j < J + N$ and also, because nothing would be changed thereby, for those $j$ for which $\hat{R}_{bj} = 0$ and those $i$ for which $\hat{R}_{ip} = 0$. 
3.6. The Computer Routine for Quadratic Programming

I have prepared a computer routine for quadratic programming which implements the Simplicial Algorithm as described here. Appendix C describes the technical details of this routine, shows how to use it, and presents the machine instructions in the FORTRAN language.

The main features of the routine are as follows. Unless modified appropriately, it requires that \( J + N \leq 105 \). Input data can be prepared in nearly any form suitable for the user, since it will accept variable format indications. Output is in a fixed format in which the optimal values of only the basic variables are given, and these are segregated into program variables and slack variables, and indexed in order. Also, a summary of the computations is given, indicating for example the number of iterations required. Any number of problems can be solved in sequence. As presently constituted, the routine requires an initial feasible basis identification as input, and no provision is made for checking that the \( A \) matrix is negative semidefinite.

The routine has been tested on approximately 120 test problems, of which

1. 3 for which \( J + N = 4 \) were checked by hand,

2. 56 for which \( J = 12, N = 1 \), were portfolio analysis problems prepared from actual data, and

3. about 60 for which \( J + N \geq 50 \) were prepared using a random number generator assuming \( A \) to be a diagonal matrix.

For the most part, the routine is self-checking, since if the arithmetic is correct then finding all nonnegative values for the basic variables assures a correct solution.
None of the test problems have required as many as \( J + N \) iterations. For each iteration, the pivoting operation requires \( (J + N)^2 \) multiplications, so a conservative estimate of the number of multiplications required is \( (J + N)^3 \) for the pivoting operations, \( (J + N)^2/2 \) for determining critical values, and if an initial basis which is not all slacks is specified \( (J + N)^3/2 \) for obtaining the initial basic representation (cf. Orden). Allowing a factor of 2 for bookkeeping operations, the estimated calculation time is therefore proportional to

\[
[3(J + N)^3 + (J + N)^2] .
\]

On an IBM 7090 computer, for which the multiplication time is about 25 microseconds, 75 seconds would be a conservative estimate of the calculation time required for a problem with \( J + N = 100 \). The input-output time would be on the order of 3 to 5 seconds. Actually, running times have been an order of magnitude less than this; viz., five to twenty seconds.

It should be mentioned here that linear programming problems, although solvable by this computer routine, are more efficiently solved by specially designed routines modeled on Dantzig's Simplex Method, which takes advantage of the special structure of the problem to condense the data storage requirements and to eliminate redundant arithmetical operations.

3.7. Other Algorithms for Quadratic Programming

Several other algorithms are available for quadratic programming. Among the authors who have proposed algorithms are Beale, Hildreth, Houthakker, Lemke, Markowitz, Theil and Van de Panne, and Wolfe (1959). Most of this work is reviewed in the surveys of Wolfe (1962) and Dorn (1965).
Only Wolfe's and Beale's algorithms have been implemented in computer routines, and the computer routine for Wolfe's algorithm is the only one available for general use (on the IBM 704 and 709 series of machines).*

The Simplicial Algorithm for quadratic programming proposed here is very similar to Beale's algorithm; indeed, it can be shown that starting from the same initial basis they will proceed through the same sequence of bases to the solution. Nevertheless, Beale's theoretical justification differs substantially, and his definition of a basis is restrictive. Also, he is forced to execute pivoting operations in two parallel tableaux.†

In perspective, the Simplicial Algorithm for quadratic programming can be construed as an extension of Beale's algorithm in which the use of the floating pivot variable and the concept of a semibasis make possible simpler and easier pivoting operations on a more compact tableau.

Wolfe's algorithm consists essentially of exploding the quadratic programming problem into a larger linear programming problem. This produces a tableau only slightly larger than the one here, and the computations required in each iteration are about the same. However, Wolfe says his algorithm requires an average of 4(J + M) iterations, whereas experience indicates that (J + M) is a conservative estimate for the number of iterations required by the Simplicial Algorithm. This is to be expected, since Wolfe loses considerable information about the structure of the problem when he converts it into a linear programming problem. However, a direct

* In remarks before the University of Chicago Symposium of Mathematical Programming, June 20, 1962, Dr. Wolfe said that Beale's algorithm appears to be the most efficient in terms of computer time among those algorithms mentioned above. However, Dorn asserts that Beale's algorithm (as well as Lemke's and Theil and Van de Panne's) will not work for objective functions that are not negative definite.

† Markowitz's algorithm, which requires that the objective be negative definite, is specially designed for portfolio analysis, and has been used in routines for that purpose.
comparison between the two computer routines on identical problems has not been made as yet.
4. Concave Programming

Algebraic techniques are my concern in this section also. Here I extend the usefulness of the techniques developed for quadratic programming in Section 3 to more general concave programming by employing the technique of successive approximations.

In the special case of quadratic programming, linearity of the equations (2-6) in the Major Theorem permits refinement of the algebraic technique to a sequence of simple pivoting operations. In the general case, the difficulty of solving simultaneous nonlinear equations drastically alters the economics of computational efficiency. Nevertheless, by judiciously combining familiar approximation methods with the simplicial procedure, a relatively easy algorithm is still possible which differs in only a few respects from the computational scheme for quadratic programming.

4.1. Introduction

Quadratic programming is easy because linear equations obtain in the conditions for a solution. In fact, linear algebraic operations are virtually the only kind that are even feasible in large problems: when nonlinear equations occur the only recourse is to solve them by successive linear approximations.

In our case, quadratic functions in the programming problem yield linear equations in the conditions for a solution. Our job, therefore, is to construct a method for solving general problems by solving a succession of approximating quadratic problems.

A systematic procedure for this purpose, however, must be developed with computational efficiency a primary consideration, and a fact of the
situation is that complete reapproximations are burdensome and not very rewarding after the neighborhood of the solution has been reached. Consequently, we shall consider a modified approach in which only the linear fit of the approximation is updated after the first few times, but not the quadratic fit.

In the following discussions, the technique of quadratic approximations is reviewed and some notation is established, and then the computational procedure is described. Remarks on the conditions under which the approximations will converge are deferred until last. Sections 5 and 6 describe applications of the method in two special types of problems.

4.2. The Quadratic Approximation

According to the Equivalence Theorem of Kuhn-Tucker presented in Section 2, the concave programming problem is equivalent to a saddle problem with objective function

$$G(x; u) = g(x) + u^T f(x).$$  \hspace{1cm} (4-1)

Here the objective and constraint functions are not disjoint, and it is meaningful to approximate the problem by approximating \( G(x; u) \) by another function. When we refer to an approximating problem, therefore, we shall mean that the objective function of the saddle problem has been approximated. In the following discussions it will always be assumed that \( g(x) \), and each \( f_n(x) \), are twice differentiable.

Recall from the introductory remarks above that we must consider successive quadratic approximations which are updated only in the linear fit after an initial (or perhaps several) quadratic fit. The situation will be as follows. Initially we will obtain a complete quadratic approximation
to \( G(x; u) \) at a point \((x^*; u^*)\); thereafter, we will update the linear fit of the approximation at a new point \((x^k; u^k)\) for iteration \(k = 2, 3, \ldots \) (by convention \((x^0; u^0) = (x^1; u^1)\)).

The **quadratic approximation** to \( G(x; u) \) for a first-order fit at \((x^k; u^k)\), and a second-order fit at \((x^*; u^*)\), is a function \( a_Q^k(x; u) \) of the same form as \( G \) would take for a quadratic programming problem, viz.,

\[
a_Q^k(x; u) = y^t x + x^t Ax + u^t [c - Dx] + \text{Constant}, \tag{4-2}
\]
such that

(a) the first partial derivatives of \( G \) and \( a_Q^k \) are the same at \((x^k; u^k)\), and

(b) the second partial derivatives are the same at \((x^*; u^*)\).

Differentiating in \((4-1)\) and \((4-2)\), we see that this definition requires the following equalities:†

(a) **First-Order Conditions**, \( \nabla G(x^k; u^k) = \nabla a_Q^k(x^k; u^k) \); viz.,

\[
(1) \quad p + (A + A^t) x^k - D^t u^k = \nabla g(x^k) + \nabla f(x^k) u^k ;
(11) \quad c - Dx^k = f(x^k) ;
\]

(b) **Second-Order Conditions**, \( \nabla^2 G(x^*; u^*) = \nabla^2 a_Q^k(x^*; u^*) \); viz.,

\[
(1) \quad (A + A^t) = \nabla^2 g(x^*) + \sum_n u_n \nabla^2 f_n(x^*) ;
(11) \quad -D^t = \nabla f(x^*) . \tag{4-3}
\]

The second-order conditions determine the matrices \( A \) and \( D \), and then with these in hand one can determine the vectors \( p \) and \( c \) from the first-order conditions.

---

† The iterated gradient operator \( \nabla^2 \) is defined to yield the symmetric matrix of second partial derivatives of a scalar valued function; e.g., \( \nabla^2 g(x) = (\partial^2 g(x)/\partial x_i \partial x_j) \).
We shall define $\mathbf{R}^k$ here just as we defined $\mathbf{R}$ in Section 3, except that we will append a $(J + H)$-dimensional identity matrix on the right for use in the approximating procedure:

$$
\mathbf{R}^k = \begin{bmatrix}
A + A^t & 0 & I_J & -D^t & -p & I_J & 0 \\
D & I_N & 0 & 0 & c & 0 & I_N
\end{bmatrix}, \quad (4-4)
$$

where $A$, $D$, $p$, and $c$ are all determined by (4-3). Segregating the basic and nonbasic columns, this can be formulated as

$$
\mathbf{R}^k = \begin{bmatrix}
\mathbf{R}^k_B & \mathbf{R}^k_B & \mathbf{R}^k & \mathbf{I}
\end{bmatrix}, \quad (4-5)
$$

in obvious notation. The basic representation is then

$$
\mathbf{R}^k = \begin{bmatrix}
I & \mathbf{R}^{-1} & \mathbf{R}^{-1} & \mathbf{R}^{-1} & \mathbf{R}^{-1} & \mathbf{R}^{-1}
\end{bmatrix}, \quad (4-6)
$$

4.3. The Approximating Procedure

Depending upon whether or not a new second-order fit is obtained, we shall say that a major or minor reapproximation is made. In general, our methodology will be that for the first few times (until we reach the vicinity of the solution) we will make major reapproximations, and thereafter, only minor reapproximations.

The complete procedure for a major reapproximation is to take the existing values of the program variables (viz., an initial guess $(x^*; u^*)$, or whatever improved values may have been obtained in the course of the algorithm) and obtain a new representation $\mathbf{R}^1$ by calculating $A$, $D$, $p$, and $c$ via (4-3).

The procedure for a minor reapproximation is much easier. On hand in the course of the algorithm will be the current basic representation $\mathbf{R}^{k-1}$ and we must calculate $\mathbf{R}^k$. Only column $L = 2(J + H)$ of changes in a minor reapproximation, however, and from (4-6) the formula for the new values in this column is
\[ \frac{s_{L}^{k}}{s_{B}^{k}} = R_{B}^{k-1} \cdot s_{R}^{k}, \quad (4.7) \]

where \( L \) denotes the column index. Hence, we obtain the new values by calculating \( s_{R}^{k} \) and multiplying it times the matrix existing in the last \((J + N)\) columns of \( s_{R}^{k-1} \). Note that

\[ s_{R}^{k} = \begin{bmatrix} -P \\ c \end{bmatrix}, \quad (4.8) \]

is obtained from (4.3) using as \((x^{k}; u^{k})\) the values of the primal and dual program variables determined currently in the course of the algorithm, and using \((A + A^t)\) and \( D \) as determined earlier in the algorithm at the last \((x^*; u^*)\) at which a major reaproximation was made.

4.4. The Extended Simplicial Algorithm

In its purest form the algorithm consists merely of alternating between the approximating procedure and the Simplicial Algorithm for quadratic programming, the substance of the procedure being in each cycle to solve the quadratic problem obtained by a major reaproximation.

This form of the algorithm is inefficient because in each cycle it throws away a major investment consisting of an (approximately) optimal basis, and for that basis, a basic representation already calculated. It is to preserve this investment that we use minor reaproximations: even though convergence must then be slower in terms of the number of successive approximations, each reaproximation is much easier. It is also to preserve this investment that after a minor reaproximation we will attempt to use the existing basis and basic representation as the starting point in the Simplicial Algorithm.
4.5. Primal Infeasibility

Two recourses are open to us should we ever lose primal feasibility during the approximation procedure. The one which certainly will work is to return to Step 2 and start over either from the initial basis or from a redetermined initial basis which is feasible for the current approximation. This way, however, is laborious, so we will resort to it only if the easier method fails.

The other method, the easier one, is to modify the Simplex Algorithm so that if any primal or dual basic variable is negative, then it will be driven to zero using the simplex procedures. One change in the simplex procedures, however, is necessary in the case of primal infeasibility. The second critical value \( z_p \) of the parametric variable, that value at which it is no longer "feasible" to increase the parametric variable further, is determined as the first value at which any variable which has been positive becomes negative. Although a proof that this method will always work has not been obtained, in the general case of primal or dual infeasibility it has the rationale that by systematically driving negative variables to zero, without letting any other variable become negative, we will eventually eliminate all negative variables.

In any case, when primal infeasibility occurs, this extension of the simplex procedures is mandatory, because then it is possible that the values of the basic primal variables are determined by the relation 
\[-f(x) + s = 0 \] in (2-6) independently of the values of the dual variables.

\[\text{I conjecture that this change is desirable also for dual infeasibility.}\]

\[\text{Over a hundred test problems of large size, } J + N > 50, \text{ have been solved without ever failing to maintain primal feasibility by this method. However, my computer routines have provisions in them to start over from the initial basis if this method does not regain primal feasibility in } J + N \text{ iterations of the simplex procedures.}\]
Consequently, increasing the nonbasic dual variable which is complementary to the negative primal variable will not suffice to drive it to zero unless we allow semibases to occur for which a nonbasic primal variable is the parametric variable. This is done by allowing the non-basic dual variable to replace a basic dual variable which becomes negative.

4.5. Efficiency and Convergence

A quadratic approximation has at least one thing to recommend it: using the approximating quadratic problem one can search easily for an approximately optimal basis, giving a significant headstart in the Simplicial Algorithm discussed in Section 2. Whether or not it is worthwhile to continue with successive quadratic approximations might seem to be in doubt were it not that computational experience to be reported in Sections 5 and 6 indicates that, besides being the easiest thing to do, it is also efficient. In addition, maintaining primal feasibility by the modification of the Simplicial Algorithm discussed above has been successful.

Convergence of the approximating procedure has not been found to be a serious problem in practice from a computational point of view: first, because in practice it is usually easy to make an initial guess \((x^*, u^*)\) which is at least within an order of magnitude of the optimal values; and second, experience and theory both indicate that, at least in the types of (nonpathological) problems of primary interest, the neighborhood of the optimal values in which the procedure will converge is relatively large. The first point is idealized in the problem of linear programming under uncertainty to be discussed in Section 5, for which the
structure of the problem permits one to make a very good guess from a simple rule of thumb.

The theoretical evidence that the neighborhood of convergence is usually large is based upon the presumption that $\nabla^2 g$, $\nabla^2 f_n$, and $\nabla f$ do not change radically in the neighborhood of an approximation. The argument can be presented briefly as follows. A quadratic approximation is equivalent to approximating the functions in the equation (2-6) linearly. If we represent these equations abstractly as $h(x^0) = 0$, then in the $k$-th iteration the approximation procedure amounts to approximating $h(x)$ by

$$ wh^k(x) = [h(x^k) - \nabla h(x^*)^t x^k] + \nabla h(x^*)^t x. \quad (4-9) $$

Hochstrasser shows that successive approximations of this type, without allowing for the simplicial procedures, will converge if $x^*$ and all $x^k$ are in a neighborhood of $x^0$ for which all of the eigenvalues of

$$ I - [\nabla h(x^*)^t]^{-1} \nabla h(x) \quad (4-10) $$

are less than unity in absolute value for every $x$ in the neighborhood. As an approximate condition we have that all of the eigenvalues of

$$ I - [\nabla h(x^*)^t]^{-1} \nabla h(x^0) \quad (4-11) $$

must be less than unity in absolute value. Certainly convergence is assured if we choose $x^* = x^0$, because (4-11) will then be of the form $I - I$ to yield the zero matrix, and there will be a neighborhood of $x^0$ in which convergence is assured. If $\nabla h$ is not greatly variable, then this neighborhood will be large; the ultimate occurs when $h$ is linear (quadratic programming), for then $\nabla h$ is constant and (4-11) reduces to the zero matrix. Equivalently, the neighborhood is large if $\nabla^2 g$, $\nabla^2 f_n$, and $\nabla f_n$, all of which enter into the composition of $\nabla h$, are not greatly variable.
It should be noted that limitations of the algorithm as regards convergence are inherent in dealing with nonlinear equations. The limitations of the theory and methods of solving nonlinear equations are an impasse to which one working in the theory and methods of programming must be reconciled. Although the computer routines described in Sections 5 and 6 have never failed to converge for test problems, it must be admitted that should they ever fail, the only recourse is to start over with another (better) initial guess.

4.6. Other Methods

Recent surveys by Wolfe (1962) and Dorn (1963) describe the other methods that have been proposed for general concave programming. Of these, the most prominent are the gradient-projection method of Rosen, the cutting-plane method of Kelley, and the method of feasible directions of Zoutendijk. Only the first of these has been implemented in a computer routine, and then only for linear constraints: the routine is not available for general use. Comparisons with the method proposed here are not possible at present, at least in terms of computational efficiency. It should be mentioned here, however, that Kelley's method will not work for nonlinear constraints, mainly because in effect he omits the terms $\sum_n u_n \nabla^2 f_n(x^*)$ in (4-5).

In industrial practice, concave programming problems have usually been solved by the laborious process of approximating nonlinear functions by piecewise linear functions (cf. Miller, and also the comments of Dorn).
5. A Linear Programming Problem Under Uncertainty

A special case of concave programming which is of significant interest is a problem of linear programming under uncertainty. The problem is formulated in this section, and cast into the computational format of the Extended Simplicial Algorithm. It serves also as a vehicle to demonstrate precisely how the Extended Simplicial Algorithm is implemented in a computer routine.

5.1. Formulation of the Problem†

The problem that we shall consider is a generalized inventory stocking problem, of the type commonly known as the continuous "newsboy problem". In the simplest case there is only one product and a decision must be made as to how much inventory, say \( y \), of this product is to be stocked. The demand for this product is not known with certainty at the time of the stocking decision, but we consider its probability distribution to be known: using a tilde to denote a random variable, \( \tilde{d} \) will be the random variable of demand. ‡‡

The economic structure of the problem in this simplest case is given by a penalty function, \( p(\Delta) \), which specifies the loss incurred when a discrepancy \( \Delta = d - y \) obtains between actual demand and available inventory. For example, when demand exceeds supply the opportunity loss consists of lost sales, and goodwill; also, when the inventory stock exceeds demand there will be an inventory carrying charge on the excess. In this form,

† Several formulations of problems of linear programming under uncertainty are extant, among which are the ones of Dantzig (1955), and Charnes and Cooper. The formulation here differs substantially from both of these, however.

‡‡ We will assume throughout that \( y \) is a real-valued variable, and that \( \tilde{d} \) has a continuous probability density function.
the problem is to minimize $E[P(\tilde{d} - y)]$ for $y \geq 0$, where $E$ is the expectation operator over the distribution of $\tilde{d}$. When $P$ represents a V-shaped proportional loss structure,

$$P(d - y) = \alpha \max(0, d - y) + \beta \max(0, y - d),$$

where $\alpha$ is the unit underage cost and $\beta$ is the unit overage cost, a well-known result (cf. Schlaifer, Chapter 4) is that the optimal choice $y^o$ is $y^o = \max(0, \tilde{y}^o)$ for $\tilde{y}^o$ the solution to

$$\text{Prob}(d < y^o) = \alpha/[(\alpha + \beta)].$$

Here we shall generalize the economic structure of this problem to include multiple products, and we shall include an analysis of the programming problem of finding the best way in which to produce the vector of amounts $y^o$ required for inventory. Our interest, however, will be confined to the computational problems of solution, and will not enter into the statistical problems. In the more general inventory problem, we will assume the production and economic structures to be linear, as in linear programming, and confine ourselves to a one-stage problem.

The production structure consists of a set of activities indexed by $j$ and we will let $x$ denote the $J$-dimensional vector of activity levels to be chosen. Then besides $x \geq 0$, the choice of $x$ is restricted by $M$ linear constraints $Ax \leq c$. The output from the production activities is given by a linear production function $Ax$, where $A$ is an $I$ by $J$ matrix of production coefficients. Of this amount, an $I$-dimensional vector $y$ will be selected for inventory and the remainder thrown away, so that $0 \leq y \leq Ax$.

The economic structure consists of a $J$-dimensional vector $q$ of costs per unit level of the activities, and an $I$-dimensional vector $p$ of
prices for the products. Thus when a program $x$ of activity levels is chosen and a vector $y$ of outputs is put into inventory, the net revenue is

$$E_i p_i \min(d_i, y_i) - q^t x = p^t y - q^t x - E_i p_i \max(0, y_i - d_i).$$

The term $E_i p_i \max(0, y_i - d_i)$ represents an adjustment for discrepancies between demands and stocking levels. Additional adjustments might be necessary to reflect lost goodwill, inventory charges, etc.; we summarize all the adjustments in a penalty function $P(\Delta)$ for $\Delta = d - y$. Letting $E$ be the expectation operator over the distribution of $\tilde{d}$, the expected penalty is given by $EP(\tilde{d} - y)$.

Hence, we have a decision problem in which the decision maker has to make two sets of choices: (1) the production levels $x$, (2) the inventory levels $y$. This defines

**A Linear Programming Problem Under Uncertainty**: find vectors $x$ and $y$ yielding the maximal value of

$$g(x, y) = p^t y - q^t x - EP(\tilde{d} - y)$$

subject to $x \geq 0$, $y \geq 0$,

$$y \leq Ax,$$

and

$$Dx \leq c,$$

where the penalty function $P$ is convex.

We assume that the (conditional) penalty function is convex so that the objective will be concave, making this problem a special case of the concave programming problem. In addition, in order to satisfy the differentiability conditions assumed in Section 4, we assume that the random
vector \( \mathbf{d} \) of demands has a continuous density function. We shall refer to this problem as LPUU in the discussions below.

After transforming the data of the LPUU problem into the format of the Extended Simplicial Algorithm, we will consider in detail the special case in which \( P \) represents proportional loss structures for the various products. Assuming these structures to be independent makes it possible to use only the marginal probability distributions of the unknown demands.

5.2. The Extended Simplicial Algorithm for Linear Programming Under Uncertainty

In order to transform the data of the LPUU problem into the format of the Extended Simplicial Algorithm, we need to specify the matrix \( \mathbb{R}^k \) which is the principal datum in the algorithm. Care must be exercised, however, because the notation that we have adopted for the LPUU is not consistent with the notation of Section 4. Also, there are two primal program vectors (\( x \) and \( y \)) to be determined and our notation must keep them separated.

Consider first the constraints. They are linear, and in the canonical form of Section 4 they become,

\[
\begin{bmatrix}
-A & I \\
D & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
\leq
\begin{bmatrix}
0 \\
c
\end{bmatrix}
\]  

(5-1)

Because the second-order partial derivatives of the constraint function are zero, a quadratic approximation to only the objective function suffices in the algorithm, and the data of (5-1) enter \( \mathbb{R}^k \) just as they would in a quadratic programming problem.
For the quadratic approximation to the objective function we calculate the following first and second order partial derivatives:
\[
\begin{align*}
\nabla_x g(x; y) &= -q ; \\
\nabla_y g(x; y) &= p + \nabla \text{EP}(\tilde{d} - y) ; \\
\nabla_{xx}^2 g(x; y) &= 0 ; \\
\nabla_{yy}^2 g(x; y) &= -\nabla^2 \text{EP}(\tilde{d} - y) ; \\
\n\nabla_{xy}^2 g(x; y) &= 0.
\end{align*}
\]

Hence, in the notation of (4-5)
\[
\mathbf{w}^k = \begin{bmatrix} q \\ s^k \\ 0 \\ c \end{bmatrix},
\]

where the 1-dimensional vector \( w^k \) is defined by interpreting in (4-3) and (4-4):
\[
\mathbf{w}^k = -p - \nabla \text{EP}(\tilde{d} - y^k) - \nabla^2 \text{EP}(\tilde{d} - y*) y^k.
\]

In terms of the discussion in Section 4, (5-4) reflects a quadratic approximation to the nonlinear term \( \text{EP}(\tilde{d} - y) \) in the objective function, for which a full second-order fit was last obtained at \( y* \) and which has been updated since then to a first-order fit at \( y^k \).

The full tableau (4-4), with the columns appropriately labeled, is as follows, using (5-1) and (5-2) and letting (5-4) define \( w^k \):
\[
\mathbf{R}_k = \begin{bmatrix} x \\ y \\ s^h \\ s^c \\ v^x \\ v^y \\ u^h \\ u^c \\ 0 \\ 0 \\ 0 \\ 0 \\ I^0_x \\ 0 \\ A^t \\ -B^t \end{bmatrix}, \quad q \iota \\ I^0_y \\ 0 \\ 0 \\ 0 \\ 0 \\ I^0 \iota \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ c^k \iota \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ I^0 \iota \\
\begin{bmatrix}
0 \\
\nabla^2 \text{EP}(\tilde{d} y^k) \\
-A \\
-b \\
\end{bmatrix}
\]

With \( \mathbf{R}_k \) in hand there is little more to be said beyond the instructions for the algorithm given in Section 4: (1) a major reapproximation is accomplished by recalculating a new tableau (5-5); (2) the Simplicial
Algorithm, performed on the basic representation \( \mathbf{w}_R^k \) obtained from (5-5), is as described in Section 3; (3) a minor reapproximation amounts to using the current solution for the optimal value of \( y \) as \( y^k \) in (5-5) to determine \( \mathbf{w}_R^k \), and then multiplying

\[
\begin{bmatrix}
q \\
\mathbf{w}_R^k \\
0 \\
c
\end{bmatrix}
\]

(5-6)
times the inverse of the basic matrix (stored in the last \( J + N \) columns of \( \mathbf{w}_R^k \)) to get the new value of column \( L = 2(j + N) + 1 \) in \( \mathbf{w}_R^k \).

5.3. Proportional Losses

In a special case of practical interest the penalty function \( P \) reflects a proportional, or \( V \)-shaped, loss structure. For example, if demand exceeds production, then the penalty, representing lost sales and goodwill, might be proportional to the discrepancy by a factor \( \alpha \). Similarly, when production exceeds demand the cost of disposal or the inventory carrying charge might be proportional to the overage by a factor \( \beta \).

In the case of several products indexed by \( i = 1, 2, \ldots, I \), we consider a penalty function reflecting independent proportional loss structures for the various products:

\[
P(d - y) \equiv \sum_i \left( \alpha_i \max(0, d_i - y_i) + \beta_i \max(0, y_i - d_i) \right).
\]

(5-7)

For this function,

\[
\partial \mathbb{E}(d - y) / \partial y_i = -\alpha_i \ \text{Prob}(d_i > y_i) + \beta_i \ \text{Prob}(d_i \leq y_i),
\]

(5-8)

and,

\[
\partial^2 \mathbb{E}(d - y) / \partial y_i \partial y_k = \begin{cases} 
0 & \text{if } i \neq k \\
[\alpha_i + \beta_i] \ \text{Prob}(d_i = y_i) & \text{otherwise}
\end{cases}
\]

(5-9)
5.4. **The Computer Program for LPUU with Proportional Losses**

A computer routine has been prepared which implements the Extended Simplicial Algorithm for LPUU with proportional losses. Currently the routine is prepared to handle only Normal probability distributions; however, slight changes will enable it to handle any other tabulated distribution. Appendix C describes the technical features of this routine, shows how to use it, and presents the machine instructions in the FORTRAN language.

The main features of the routine are as follows. Without modification it will accept problems for which \( J \leq 50, \ I \leq 50, \ N \leq 30 \), provided \( J + 2I + N \leq 80 \). Variable format provisions allow the user to prepare input data in nearly any form that is convenient. Output is in a fixed format in which the optimal values of the basic variables are given in order, but segregated by program and slack variables. A summary of the computations follows which gives the number of successive approximations required to obtain a degree of accuracy specified by the user, and also the number of pivoting operations required. Any number of problems can be solved in sequence. As presently written, the routine requires an initial feasible basis as an input datum. A rather complete set of error indications is built into the routine, including checks that the objective is concave \((c_i + \beta_i \geq 0 \text{ for each } i)\). Also, a maximum number of allowable approximations can be specified by the user; e.g., to halt diverging approximations. Provision is also made for the user to specify the number of major re-approximations (complete quadratic fits) to be made.

It is not necessary to make an initial guess \( y^* \) for this routine: as presently written the routine automatically takes \( y_1^* \) as the sum of
the mean and standard deviation of $\bar{d}_1$. The reason for this is that among
the conditions (cf. (5-5)) for a solution we find that
\begin{equation}
\hat{u}_1 = \alpha_1 - [\alpha_1 + \beta_1] \text{Prob}(\bar{d}_1 < y_1)
\end{equation}
when $y_1$ is not constrained to zero. Now assuming $\alpha_1$ to be larger than
$\beta_1$ and $\hat{u}_1$ to be an order of magnitude smaller than $\alpha_1$, (5-10) in the
form
\begin{equation}
\text{Prob}(\bar{d}_1 < y_1) = \frac{\alpha_1 - \hat{u}_1}{\alpha_1 + \beta_1}
\end{equation}
is approximately satisfied by choosing $y_1$ as described above.

The computer routine has been tested on approximately 70 test prob-
lems, of which

1) 1 for which $J = 3$, $I = 1$, $N = 2$,
2) for which $J = 3$, $I = 2$, $N = 1$, and
3) for which $J = 10$, $I = 5$, $N = 5$,
were checked by hand, and

2) over 60 for which $J \geq 10$, $I \geq 5$, $N \geq 5$, among which were
problems reaching the capacity of the routine, were prepared
using a random number generator.

The routine is largely self-checking, since finding all nonnegative values
for the basic variables assures a correct solution provided the arithmetic
has been correct. However, no measures have been taken to guard against
deceptive appearances of convergence in the approximating procedure: the
analysis of convergence has not been carried far enough to ascertain what
pathological possibilities exist in this regard.
None of the test problems have required as many as \((J + 2I + N)\) pivoting operations. The total number of approximations required, however, has varied widely depending upon the number of major reapproximations specified. An example will suffice to illustrate the situation: for a problem with \(J = 10\), \(I = 5\), \(N = 5\), and the convergence criterion that the maximum difference between the values of the basic variables in successive approximations be less than \(10^{-5}\), the following results were obtained in successive tests:

<table>
<thead>
<tr>
<th>Number of Major Reapproximations</th>
<th>Number of Minor Reapproximations</th>
<th>Total Number of Approximations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

(The number of major reapproximations includes the initial approximation.)

The indications are, therefore, that selection of the number of major reapproximations plays an important role in determining computational efficiency. An "optimal" tradeoff between major and minor reapproximations has not been investigated, although 2 for small problems and 3 for large problems seems a good rule of thumb.
6. Variable-Factor Programming

Another special case of concave programming to which the Extended Simplicial Algorithm can be applied is variable-factor programming, which is of significant interest in economic planning and in industrial scheduling. In this section I formulate the problem of variable-factor programming, and show how the Extended Simplicial Algorithm is implemented to solve the problem in a computer routine.

6.1. Formulation

In variable-factor programming we consider situations in which both variable and fixed factors of production play a role, and in which not only a program vector of activity levels, but also a program of variable-factor inputs must be decided upon. Agricultural planning will be the context of the formulation here, but another important context is scheduling of oil refinery operations (cf. Manne).

Let \( x = (x_j) \) be the \( J \)-dimensional program vector of activity levels, specifying the acreages to be planted in various crops indexed by \( j \). Besides the natural constraint \( x \geq 0 \), limitations of land available and of other fixed factors of production enforce \( M \) fixed-factor constraints \( D x \leq c \).

Now the net yield (in monetary units) from an acre of crop \( j \) depends upon the \( M \)-dimensional vector \( z^j = (z^j_m) \) of variable-factor amounts allocated to it, where \( m \) indexes the variable factors of production such as man-hours of cultivation, fertilizer, and irrigation water. Letting this dependency be expressed by a yield function \( y_j(z^j) \), the total yield from the agricultural program is given by \( \Sigma_j x_j \ y_j(z^j) \).
Finally, letting \( b \) be the \( M \)-dimensional vector of variable-factor supplies, the problem can be defined as follows:

**Variable-Factor Programming:** find vectors \( x \) and \( z^j \), \( j = 1, \ldots, J \), yielding the maximal value of 

\[
g(x, z) = \sum_j x_j y_j(z^j)
\]

subject to \( x \geq 0, z^j \geq 0, \)

\[
Dx \leq c,
\]

and

\[
\sum_j x_j z^j \leq b.
\]

This problem will be referred to as the \( VFP \) problem in the discussions below.

We will assume that each of the functions \( y_j(z^j) \) is concave; nevertheless, the variable-factor constraints are not concave, so this problem is not a special case of the concave programming problem. Even so, we know from the theorems of Kuhn and Tucker that any solution to the saddle problem is a solution to the programming problem even without concavity,\(^\dagger\) and in the following discussions we shall find that we can obtain a solution to the saddle problem by finding a solution to a certain problem (called the dual problem) which is a concave programming problem.

\(^\dagger\) As mentioned in Section 2, p. 11, the concavity assumption was not needed by Kuhn and Tucker in order to show that any solution \((x^*; u^*)\) to the saddle problem with

\[
G(x; u) = g(x) + u^T f(x)
\]

yields \( x^* \) as a solution to the programming problem.
6.2. Reduction of the Problem Via the Saddle Problem

A characteristic feature of the VFP problem is the large number of variables involved; consequently, it is desirable to reduce the problem by a preliminary transformation derived from the saddle problem.

For variable-factor programming the objective function in the saddle problem (cf. Section 2) takes the form,

\[ g(x, z; u, v) = \sum_j x_j [y_j(z^j) - v^t z^j] + u^t [c - Dz] + v^t b, \tag{6-1} \]

and we see that given a value of \( v \) we can maximize for the \( z^j \) independently of the other variables. Hence, defining new functions

\[ \tilde{y}_j(v) = \max_{z_j \geq 0} [y_j(z^j) - v^t z^j], \tag{6-2} \]

we obtain a simpler saddle problem with objective function,

\[ \tilde{g}(x; u, v) = x^t [\tilde{y}(v) - y^t u] + u^t c + v^t b, \tag{6-3} \]

where \( \tilde{y} \) is the vector of functions (6-2). But this is just the saddle problem of another programming problem in which the role of the primal and dual variables are reversed; viz., the

**Dual VFP Problem:** find vectors \( u \) and \( v \) yielding the minimal value of

\[ u^t c + v^t b \]

subject to \( u \geq 0, v \geq 0, \) and

\[ \tilde{y}(v) \leq D^t u. \]

By eliminating the variable-factor programs \( z^j \) from independent consideration, the dual VFP problem reduces the computational burden and the data storage requirements substantially.
In order to solve the dual VFP problem using the Extended Simplicial Algorithm, it is necessary that the nonlinear functions $\bar{y}(v) = \bar{y}_j(v)$ be convex: it can be verified from (6-2) that $\bar{y}_j(v)$ will be convex if $y_j(x^j)$ is concave.

Retracing our steps from the dual VFP problem to the primal VFP problem, we can establish that a solution to the dual problem yields a solution to the primal problem as follows. Given a solution $(u^o, v^o)$ to the dual problem, the Equivalence Theorem shows that there must exist a vector (of dual variables) $x^o$ such that $(x^o; u^o, v^o)$ solves the saddle problem with objective function $\bar{G}$ in (6-3), and in the course of the computations we can actually obtain $x^o$. Now since $\bar{G}$ was obtained by maximizing $G$ in (5-1) with respect to $x^j$, there will exist a matrix $z^o = (z^i_j)$ of vector values for which (6-2) is an attained maximum for $v = v^o$, and $(x^o, z^o; u^o, v^o)$ must solve the saddle problem with objective function $G$ in (6-1). Finally, we know (Section 2, p. 11, first line) that this solution must yield $(x^o, z^o)$ as a solution to the primal VFP problem.

6.3. The Extended Simplicial Algorithm for Variable-Factor Programming

As in Section 5 for linear programming under uncertainty, our job here is to transform the data of the dual VFP problem into the format of the Extended Simplicial Algorithm. For this it suffices to find the form of the tableau representation $W^k$ defined in (4-4) and (4-3). Some translation of notation is necessary also.

Referring to (6-3), we can calculate the first and second order partial derivatives needed for the approximation scheme. Those that are not identically zero are as follows:
\[ \nabla \mathbf{y} = \mathbf{D}^t \mathbf{u} ; \quad \nabla \mathbf{z} = \mathbf{D} \mathbf{x} ; \quad \nabla \mathbf{v} - \mathbf{b} + \nabla \tilde{y}(\mathbf{v}) \mathbf{x} ; \]
\[ \nabla^2 \mathbf{y} = -\mathbf{D} ; \quad \nabla^2 \mathbf{z} = \nabla \tilde{y}(\mathbf{v})^t ; \quad \nabla^2 \mathbf{v} = \sum_{j} x_j \nabla^2 \tilde{y}_j(\mathbf{v}) . \]

(6-4)

Turning to the formulation of the tableau, we insert the information from (6-4) into place according to (4-4) and (4-5) as follows, where the column headings are self-explanatory except that \( \mathbf{w} \) denotes the vector complementary to \( \mathbf{x} \):

\[
\begin{bmatrix}
\mathbf{u} & \mathbf{v} & \mathbf{v}^s & \mathbf{s}^u & \mathbf{s}^v & \mathbf{x} \\
0 & 0 & 0 & I_N & 0 & D \\
-\mathbf{E}_j \mathbf{x}_j & \nabla^2 \tilde{y}_j(\mathbf{v})^s & 0 & 0 & I_N & 0 \\
-\mathbf{D}^t & \nabla \tilde{y}(\mathbf{v})^s & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & I_N \\
0 & 0 & 0 & 0 & 0 & I_N \\
\end{bmatrix}
\]

(6-5)

for,

\[ \mathbf{w}_b^k = \mathbf{b} - \sum_{j} x_j \nabla^2 \tilde{y}_j(\mathbf{v})^s \mathbf{v}^k + [\nabla \tilde{y}(\mathbf{v})^k - \nabla \tilde{y}(\mathbf{v})^s] \mathbf{x}^k , \]

(6-6)

and

\[ \mathbf{w}_a^k = \tilde{y}(\mathbf{v})^k - \nabla \tilde{y}(\mathbf{v})^s \mathbf{v}^k . \]

(6-7)

This completes the formulation of the problem in terms of the Extended Simplicial Algorithm provided we know how to compute \( \nabla \tilde{y} \) and \( \nabla^2 \tilde{y} \).

6.4. Computing \( \nabla \tilde{y}_j \) and \( \nabla^2 \tilde{y}_j \)

In order to find formulas for computing \( \nabla \tilde{y}_j \) and \( \nabla^2 \tilde{y}_j \), we delve further into the relationship between the functions \( \gamma_j \) and \( \tilde{y}_j \) defined by (6-2).

Let \( z^j(v) \) be the value of \( z^j \) at which the maximum occurs in (6-2) given \( v \); then, differentiating formally in (6-2) yields

\[ \nabla \tilde{y}_j(v) = \nabla z^j(v) \nabla y_j(z^j(v)) - z^j(v) - \nabla z^j(v) v . \]

(6-8)
However, a necessary condition for the maximum in (6.2) is that
\[ \nabla y_j(z^j(v)) \leq v, \]  
with strict inequality occurring for the k-th component only if \( z_k^j \) is constrained to zero, in which case \( \nabla z_k^j(v) = 0 \). Hence, (6.8) reduces to
\[ \nabla \tilde{y}_j(v) = -z^j(v), \]  
and to calculate \( \nabla \tilde{y}_j(v) \) we must solve (6.2) for the optimal \( z^j(v) \) given \( v \).

An easy special case occurs when \( y_j \) is separable into
\[ y_j(z^j) = \sum_z y_{jm}(z^j). \]  
For, in (6.2) an attained maximum is then defined by the univariate conditions that, unless constrained to zero, \( z_m^j(v) \) is determined by
\[ \frac{\partial y_{jm}(z^j_m)}{\partial z^j_m} = v_m. \]  
Hence, letting \( h_{jm}(\cdot) \) be the inverse function \( (\partial y_{jm}/\partial z^j_m)^{-1} \),
\[ z_m^j(v) = \max(0, h_{jm}(v_m)), \]  
which yields (6.10) in an appropriate form for calculation.

Considering only the separable case (6.11), we can calculate
\[ \nabla^2 \tilde{y}_j \]  
from (6.10) and (6.12) as
\[ \frac{\partial^2 \tilde{y}_j(v)}{\partial v_k \partial v_m} = \begin{cases} 
0 & \text{if } k \neq m \\
0 & \text{if } k = m \text{ but } z_m^j(v_m) \text{ is constrained to zero} \\
\nabla h_{jm}(v_m) & \text{otherwise}
\end{cases}. \]  
From (6.12) and (6.13) it is clear that a necessary analytical step before the problem is ready for calculations is inversion of the functions \( \nabla y_{jm} \), and also finding the first derivative of the inverse function.
6.5. The Computer Routine for Variable-Factor Programming

I have prepared a computer routine which implements the Extended Simplicial Algorithm for variable-factor programming. Appendix C describes the technical details of this routine, shows how to use it, and presents the machine instructions in the FORTRAN language.

This routine has been tested sufficiently for only one type of function \( y_j(s^j) \); namely, a quadratic function

\[
y_j(s^j) = y_j \left[ 1 - q_j \left( 1 - \frac{\sum_k a_j s_j^k}{s_j^j} \right)^2 \right] - c_j ,
\]

for \( J = 11, M = 15, K = 2 \), occurring in an actual agricultural planning problem.† For this problem it was possible to check the results against answers obtained by other means.

I have, in addition, made provisions in the routine for it to handle separable functions \( y_j = \sum_k y_{jm} \) for which each \( y_{jm} \) is of one of the following types, where \( a, b, c, d \) are parameters and \( a > 0 \):

\[
\begin{align*}
&(as + b)^c + d , & 0 < c < 1 ; \\
&-(as + b)^c + d , & c < 0 ; \\
&a \log (s + b) + d ; \\
&e^{as+b} + d .
\end{align*}
\]

Although the routine appeared to solve some ten test problems prepared using a random number generator, I have not checked these results by hand calculations.

† Professor Robert Dorfman introduced me to this problem, and provided the data for the analysis.
Unless modified, the routine requires that $J \leq 50$, $N \leq 40$, $M \leq 20$, and $J + N + M \leq 80$. Otherwise, its main features are similar to the other two routines.
APPENDIX A

Solution of a Numerical Example by the Simplicial Algorithm

In this appendix a numerical example is solved using the Simplicial Algorithm described in Section 2. Although a computer routine to solve this example would apply Newton's method, as described in Section 4, the example is not solved that way here, in order to illustrate the algebraic manipulations required by the general algorithm. I am indebted to Professor John Bishop for providing this example.

Let \( J = 2 \) and \( N = 2 \), and define

\[
\begin{align*}
g(x_1, x_2) &= x_2 - (x_1 + x_2 - 1)^4 \\
f_1(x_1, x_2) &= 4x_1 - 3x_2^2 + 4 \\
f_2(x_1, x_2) &= -2x_1^2 + x_2 + 1.
\end{align*}
\] (A-1)

The simultaneous equations (2-6) take the following form:

\[
\begin{align*}
-4(x_1 + x_2 - 1)^3 + v_1 + 4u_1 - 4u_2x_1 &= 0 \\
-4(x_1 + x_2 - 1)^3 + v_2 - 6u_1x_2 + u_2 &= -1 \\
-8x_1 + 3x_2^2 + s_1 &= 4 \\
2x_1^2 - x_2 + s_2 &= 1
\end{align*}
\] (A-2)

Since the origin is feasible,

\[
\begin{align*}
f_1(0, 0) &= 4 > 0, \\
\end{align*}
\]

(A-3)

it suffices to take an initial basis for which the slack variables are all basic, viz.,

\[
B_0 = (3, 4, 5, 6).
\] (A-4)
Examining (A-2) with $x_1 = x_2 = u_1 = u_2 = 0$, the following values of the basic variables are immediate:

$$
\hat{z}_1 = 4, \quad \hat{z}_2 = 1, \quad \hat{v}_1 = -4, \quad \hat{v}_2 = -5. \quad (A-5)
$$

Choosing to drive the basic dual variable $v_2$ to zero, we calculate the critical values of its complementary nonbasic variable $x_2$ by letting $x_2$ be a parameter in (A-2). The second equation in (A-2) indicates that

$$
-4(\bar{x}_2 - 1)^3 = -1 ,
$$

or,

$$
\bar{x}_2 = 1 + \sqrt[3]{1/4}. \quad (A-6)
$$

The third and fourth equations offer the alternatives,

$$
\bar{x}_2 = \sqrt[3]{1/3}, \quad \bar{x}_2 = -1 , \quad (A-7)
$$

the second of which (being negative) is extraneous. Since $\bar{x}_2 < \bar{x}_2$ we make $s_1$ nonbasic and $x_2$ basic. This change of basis identifies $u_1$ as the new parameter to be varied in (A-2). We now have

$$
\hat{v}_2 = -1 + 4(\sqrt[3]{1/3} - 1)^3 , \quad (A-8)
$$

which is still negative, and we continue to drive $\hat{v}_2$ to zero by increasing $u_1$. From the second equation in (A-2) we see that this is accomplished for

$$
\tilde{u}_1 = [-4(\sqrt[3]{1/3} - 1)^3 + 1]/6 \sqrt[3]{1/3} . \quad (A-9)
$$

Since we are at a vertex of the feasible subspace, identified by the fact that the basic primal variables ($x_2$ and $s_2$) are completely determined by the last two equations of (A-2), we recognize that $\tilde{u}_1$ is unbounded. Hence we make $v_2$ nonbasic and $u_1$ basic, and then repeat the procedure.
The basis for the next iteration is

$$B_1 = (2, 4, 5, 7),$$ \hspace{1cm} (A-10)

and from the previous analysis we have that \( \bar{u}_1 = \tilde{u}_1 \) in (A-9). The other basic dual variable is \( v_1 \), for which the first equation of (A-2) yields

$$\tilde{v}_1 = 4([\sqrt{4/3} - 1]^5 - \theta_1] < 0.$$ \hspace{1cm} (A-11)

Observing that this is negative, we shall drive \( \tilde{v}_1 \) to zero. The critical values are obtained from (A-2) with \( s_1 = v_2 = u_2 = 0 \) and \( x_1 \) as a parameter; viz., from the first three equations

$$-4(\bar{x}_1 + \bar{x}_2 - 1)^3 + 4\bar{u}_1 = 0,$$

$$-4(\bar{x}_1 + \bar{x}_2 - 1)^3 - 6\bar{u}_1 \bar{x}_2 = -1,$$

$$-4\bar{x}_1 + 3\bar{x}_2^2 = 4,$$ \hspace{1cm} (A-12)

the first two of which reduce immediately to

$$-4(\bar{x}_1 + \bar{x}_2 - 1)^3 + 4/[4 + 6\bar{x}_2] = 0.$$ \hspace{1cm} (A-13)

It can be verified from these equations that \( \bar{x}_1 < 1/3 \). On the other hand, the last equation in (A-2) yields \( s_2 > 0 \) when \( x_1 = 1/3, x_2 = 4/5 \), satisfying the third equation. Hence, \( \bar{x}_1 < \bar{x}_2 \) and we make \( v_1 \) nonbasic to be replaced by \( x_1 \) as basic.

The basis now contains only \( u_1 \) as a basic dual variable. But from (A-12) we see that

$$4\bar{u}_1 = 1 - 6\bar{u}_1 \bar{x}_2,$$

or,

$$\bar{u}_1 = 1/[4 + 6\bar{x}_2],$$ \hspace{1cm} (A-14)

which is strictly positive. Hence, we have found the optimal basis. The optimal values of the basic variables are given by (A-2); viz., using (A-13) and (A-14),
\[ u_1^* = \frac{1}{4 + 6x_2^*} , \]

and

\[ s_1^* = 1 - 2x_1^* + x_2^* , \]

for \( x_1^* \) and \( x_2^* \) determined by the simultaneous equations,

\[-(x_1^* + x_2^* - 1)^3 + 1/[4 + 6x_2^*] = 0 , \]

\[-4x_1^* + 5x_2^* - 4 = 0 . \]

An approximate solution is given by \( x_1^* = 1/3, \ x_2^* = 4/3 \), both values being somewhat too large.
A Numerical Example of Quadratic Programming
Solved by the Simplicial Algorithm

In this appendix I solve a small quadratic programming problem using the Simplicial Algorithm as described in Section 3.

Let $J=2$, $M=2$, and in the notation of (3-2),

$$
p = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} -1 & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}, \quad c = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 2 \\ 3 & 2 \end{bmatrix}.
$$

This problem is depicted in Figure B-1. The absolute maximum of the objective function occurs at $Q=(2,3)$, and the constrained maximum occurs at $P_3=(1/2, \ 3/4)$, at which both constraints are binding. The path to the solution is indicated by the segmented line $P_0P_1P_2P_3$.

An initial feasible basis is provided by taking all the slack variables to be basic. The basic matrix is then the identity matrix, $B_0=I$, and we can skip the first step of the algorithm in which the initial basis is established, since $\hat{B}=R$.

Deleting the basic columns from $\hat{R}$ yields the tableau shown in Figure B-2, which also shows the tableaux for the successive iterations of the algorithm. The basic variables identify the rows, and the nonbasic variables the columns, according to the labeling shown. The notes on the right indicate for each tableau the information obtained from that tableau in order to identify the pivot row and column. After executing the pivoting operation four times, one obtains the solution in the last column of the fifth tableau.

The procedures all conform to the flow chart of the algorithm given in Figure 3-1. Here, we will go through these procedures for the iteration from the first to the second tableau. Looking in the last column of the first tableau for the values of the basic variables, we find that both of the dual variables are negative. Since they are both of the same magnitude we arbitrarily choose $v_1$ to be driven to zero and designate it as the floating pivot variable. Its complementary
variable is $x_1$, so we designate $x_1$ as the parametric variable. Hence, in the notation of the text, $p=1$ and we find the minimum positive ratio \( \frac{\hat{r}_{1p}}{\hat{r}_{1L}} \), where $L$ indexes the last column and $i$ runs over the rows of $v_1$ and the primal variables. These ratios are (i) for $v_1$, \((-1)/(-2)\); (ii) for $s_1$, \(2/1\); (iii) for $s_2$, \(3/3\). They are all positive and the smallest one is the one for $v_1$. Hence, we make $v_1$ nonbasic to be replaced by the parametric variable $x_1$. Executing the pivoting operation on the tableau accomplishes this basis change, and interchanging the labels on the pivot row and column prepares us for the next iteration. Because the floating pivot variable was made nonbasic in the first iteration, the second iteration starts the procedure over again. If the floating pivot variable had not been made nonbasic, then the complement of the primal variable that was made nonbasic would become the parametric variable for the second iteration. This is illustrated in the next iteration, where $u_2$ becomes the new parametric variable.
Figure B-1
### The Sequence of Tableaux

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Tableau</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>$x_1$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>1</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>$v_1$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>2</td>
<td>-1/2</td>
<td>-1/2</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>-1/2</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>5/2</td>
</tr>
<tr>
<td></td>
<td>3/2</td>
<td>7/2</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>$v_1$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>3</td>
<td>-2/7</td>
<td>1/7</td>
</tr>
<tr>
<td></td>
<td>5/7</td>
<td>1/7</td>
</tr>
<tr>
<td></td>
<td>-4/7</td>
<td>-5/7</td>
</tr>
<tr>
<td></td>
<td>3/7</td>
<td>2/7</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>$v_1$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4/8</td>
</tr>
<tr>
<td></td>
<td>-4/8</td>
<td>-7/8</td>
</tr>
<tr>
<td></td>
<td>-4/8</td>
<td>-5/8</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-2/3</td>
</tr>
</tbody>
</table>
Figure B-2 continued

<table>
<thead>
<tr>
<th></th>
<th>$v_1$</th>
<th>$s_2$</th>
<th>$v_2$</th>
<th>$s_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>-8/12</td>
<td>4/12</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6/12</td>
<td>3/12</td>
<td>3/12</td>
<td>9/12</td>
</tr>
</tbody>
</table>

$\begin{bmatrix}
0 \\
0 \\
0 \\
0
\end{bmatrix}$ $\begin{bmatrix}
6/12 \\
3/12 \\
3/12 \\
9/12
\end{bmatrix}$ $x_1$ $u_1$ $u_2$ $x_2$

Final Tableau
Appendix C

The Computer Routines

In this appendix I describe the computer routines that implement the Simplicial Algorithm for
1. Quadratic Programming,
and the Extended Simplicial Algorithm for
2. Linear Programming Under Uncertainty, and
The routines are listed separately at the end of this appendix along with sample outputs, while all of the associated material (including the source and object program card decks) that can not be included here is deposited with Professor John Bishop. The routines are available for general use, and instructions for using them are included here.

1. Preparing Input Data

The input data for each problem to be solved is prepared in a data package consisting of an identification card followed by the data on punched cards. In order to use a routine, one takes the object-program binary-card deck and appends the data packages in sequence behind it. In addition, the FORTRAN Monitor System used on most machines requires that two other cards be added: (1) a card with an * (asterisk) in column 1 and XEQ in columns 7-9, inserted before the object-program deck, and (2) a card with an * in column 1 and DATA in columns 7-10, inserted between the
object-program deck and the first data package. The entire card deck is then put on tape and mounted on logical tape unit 5 (station A2) to be read by the routine. Output is written on logical tape unit 6 (station A3) for later printing.

The identification card for each data package must be prepared in a fixed format, which differs for each routine. Because the routines allow for variable format statements, however, the other input data can be prepared in nearly any form convenient to the user, provided it is ordered properly. The user must acquaint himself with format statements.*

Exhibits C-1, 2, and 3 describe the composition of a data package for each of the three routines.

2. Organization of the Routines

The computer routines are written in the FORTRAN language.* To the person versed in this language, the routines are almost self-explanatory given an understanding of the algorithm, and of the notation being used. Consequently, the minute details of the routines are not discussed here. Exhibits C-4, 6, and 8 depict the organization of the three routines schematically in terms of relations among subroutines. Exhibits C-5, 7, and 9 then describe the functions of the various subroutines in terms of the flow charts of the algorithms; in addition, the notation in each routine

is defined in relation to the notation in the text.

3. The FORTRAN statements and Sample Output

Exhibits C-10, 12, and 14 are the FORTRAN statements for the respective routines, each organized by subroutines in the order used in Exhibits C-5, 7, and 9. I do not expect these routines to be comprehensible to every reader, but they are presented here for the person knowing the FORTRAN language, and particularly for the specialist. Also, they serve to document the work reported in the text.

Exhibits C-11, 13, and 15 are sample output sheets from the respective routines for various test problems (mostly generated from random numbers). They serve to indicate the output format, which is largely self-explanatory.
Exhibit C-1

Quadratic Programming

Data Package Composition

1. Identification

<table>
<thead>
<tr>
<th>Columns</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>J</td>
</tr>
<tr>
<td>6-10</td>
<td>N</td>
</tr>
<tr>
<td>15-18</td>
<td>EPSIL0; format E4.0: the initial basic matrix is considered singular if its determinant is less than this quantity in absolute value; e.g., l-05 means that the minimum allowable absolute value of the determinant is 10^-5.</td>
</tr>
</tbody>
</table>

(20) JEKNBD: not used, leave blank

(25) KRANNDM: indicator for random data generation, leave blank

2. The format cards, in order for data subgroups 3, 4, and 5 below.

3. Vector \( p = (p_j) \):

\[
(p_j; j = 1, \ldots, J)
\]

4. Matrix \( A = (a_{ij}) \):

\[
((a_{ij}; i = 1, \ldots, I); j = 1, \ldots, J)
\]

5. Constraint Set, \( D x \leq c \):

\[
(c_n, (a_{nj}; j = 1, \ldots, J); n = 1, \ldots, N)
\]

6. Initial basis: column \( j \neq 0 \) only if \( x_j \) is to be a basic variable, and column \((n + j) \neq 0\) only if \( u_n \) is to be basic; e.g., all zeros implies all slack variables are basic. (After 72 columns to a card, continue on next card.)
Exhibit C-2

Linear Programming Under Uncertainty

Data Package Composition

0. Before the first data package, there must be the following:
   
   1. Format card for the Normal tables:
   
   2. Parameter card with
      
      NTABLE in columns 1-5,
      DATA in columns 6-20 (format F3.3);
   
   3. Normal distribution table (upper half of the distribution);
   
   4. Normal density table (either half).

NTABLE is the number of entries per table, and DELTA is the tabulation increment.

1. Identification Card

<table>
<thead>
<tr>
<th>Columns</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>J</td>
</tr>
<tr>
<td>4-6</td>
<td>I</td>
</tr>
<tr>
<td>7-9</td>
<td>N</td>
</tr>
<tr>
<td>10-21</td>
<td>EP; cf. EPSILN, Exhibit C-1 (format E12.4; e.g., 25000-07 means that EP is 2.5 × 10^{-7}).</td>
</tr>
<tr>
<td>22-24</td>
<td>MAXIT; maximum number of approximations allowed.</td>
</tr>
<tr>
<td>25-27</td>
<td>ITMIN; number of major reapproximations required.</td>
</tr>
<tr>
<td>28-39</td>
<td>CV; accuracy desired; viz., the maximum allowable sum of squared differences between values of the basic variables in successive minor reapproximations (format E12.4; e.g., 10000-On means that CV is 10^{-n} for n ≤ 9).</td>
</tr>
</tbody>
</table>

2. The format cards, in order for data subgroups 3-8 below:

3. Vector of product prices,
   
   (p_i; i = 1, \ldots, I).
4. Vector of negatives of unit activity costs, 
   \((-d_{ij}; j = 1, \ldots, J)\).

5. Initial inventories, say 
   \((h_i; i = 1, \ldots, I)\).

6. Matrix of production coefficients, 
   \((a_{ij}; j = 1, \ldots, J); i = 1, \ldots, I)\).

7. Vector of constraint capacities, 
   \((c_n; n = 1, \ldots, N)\).

8. Matrix of the constraint set, 
   \((d_{nj}; j = 1, \ldots, J); n = 1, \ldots, N)\).

9. Parameters of the penalty function, 
   \((\alpha_1, \beta_1, \gamma_1, s_1); i = 1, \ldots, I)\) 
   where \(\bar{\gamma}_i\) and \(s_1\) are the mean and standard deviation, re- 
   spectively, of \(\gamma_i\).

10. Initial basis description: nonzero columns signify basic 
    program variables as follows,

    | Column | Program Variables |
    |--------|-------------------|
    | j      | \(x_j\) is basic  |
    | \(i + J\) | \(y_i\) is basic |
    | \(i+I+J\) | \(u^h_i\) is basic |
    | \(n+2I+J\) | \(u^c_n\) is basic |

    in the notation of Section 5.
Exhibit C-3

Variable-Factor Programming

Data Package Composition

1. Identification Card

<table>
<thead>
<tr>
<th>Columns</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>KIND = 19 for the agricultural planning problem</td>
</tr>
<tr>
<td></td>
<td>= 1 for the functions described in Section 6.</td>
</tr>
<tr>
<td>6-10</td>
<td>J; number of activities.</td>
</tr>
<tr>
<td>11-15</td>
<td>N; number of fixed-factor constraints.</td>
</tr>
<tr>
<td>16-20</td>
<td>M; number of variable-factor constraints.</td>
</tr>
<tr>
<td>21-25</td>
<td>MAXIT; maximum number of approximations allowable.</td>
</tr>
<tr>
<td>26-30</td>
<td>MINIT; number of major reapproximations desired.</td>
</tr>
<tr>
<td>40-43</td>
<td>CV; accuracy requirement; viz., maximum allowable absolute difference between values of basic variables in successive minor reapproximations (format E4.0).</td>
</tr>
<tr>
<td>50-53</td>
<td>EP; cf. EPSILN, Exhibit C-1 (format E4.0).</td>
</tr>
<tr>
<td></td>
<td>(60  KRANDM: nonzero only if random input data desired).</td>
</tr>
</tbody>
</table>

2. Format cards in order for the data subgroups 3-5 below.

3. Fixed-factor constraints,
   \[ (c^n_j; j = 1, \ldots, J); \ n = 1, \ldots, N. \]

4. Variable-factor constraints,
   \[ (b^m_m; m = 1, \ldots, M). \]

5. Initial guesses,
   \[ (v^m_m; m = 1, \ldots, M), \ (x^*_{j}; j = 1, \ldots, J). \]

6. Initial basis: a nonzero column indicates a basic variable as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>u_n</td>
</tr>
<tr>
<td>m + N</td>
<td>v_m</td>
</tr>
<tr>
<td>j + M + N</td>
<td>x_j</td>
</tr>
</tbody>
</table>
7. Parameters of the yield functions $y_j(z^j)$: each user must write a FORTRAN subroutine to calculate values (cf. Exhibit C-9) of the yield functions he is using, and this subroutine can, if necessary, read in parameter values.
Exhibit C-4
Quadratic Programming
Organization Chart of the Subroutines

QPMAIN → QPLCMN

INPUT

(input: DATACN → RANDM)

(input: NEGSD)

BASINV → BASSIN

○ PIVOT

QPLC → PIVOT

OUTPUT
### Exhibit C-5

**Quadratic Programming**

**Description of the Subroutines**

<table>
<thead>
<tr>
<th>Name of the Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPMAIN</td>
<td>This is an executive subroutine which calls the main subroutine into operation to solve successive problems until no more problems are found on the input tape, whereupon an automatic terminal exit occurs.</td>
</tr>
<tr>
<td>QPLCMN</td>
<td>This is the main subroutine, which orders the major tasks to be done. It reads in the parameters of the problem, checks them, and then calls in the functional subroutines in order.</td>
</tr>
<tr>
<td>INPUT</td>
<td>This subroutine reads in the input data and stores it in the tableau; also, the input data is written on the output tape.</td>
</tr>
<tr>
<td>BASINV</td>
<td>This subroutine obtains the basic representation for the initial basis; cf. Step 1, p. 33.</td>
</tr>
<tr>
<td>QPLC</td>
<td>This subroutine executes the simplicial procedures on the tableau of the basic representation until the optimum is found; cf. Steps 2-5, p. 33.</td>
</tr>
<tr>
<td>PIVOT</td>
<td>This subroutine executes the pivoting operation on the tableau whenever called for by BASINV or QPLC; cf. p. 30.</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>This subroutine writes the output data (solution values of the basic variables and the number of iterations required to find the optimum, on the output tape.</td>
</tr>
<tr>
<td>BASQIN</td>
<td>If the initial basis yields a singular basic matrix, then this subroutine terminates the computations.</td>
</tr>
<tr>
<td>NESSD</td>
<td>Although not completed to data, this subroutine provides an option to check that the objective is concave.</td>
</tr>
<tr>
<td>RANDM</td>
<td>These are subroutines to generate random numbers as input data, if called for by an indicator among the input parameters of the problem.</td>
</tr>
</tbody>
</table>
Exhibit C-5 continued

**Format of the Tableau**

\[
\begin{bmatrix}
 x & u & v & s \\
 A & A^t & -D^t & I_N & 0 & -p \\
 D & 0 & 0 & I_N & c
\end{bmatrix}
\]

The tableau is called \( \mathcal{W} \) in the routine, instead of \( \mathcal{R} \) and \( \hat{\mathcal{R}} \) as in the text. Additional notation is as follows:

**IB**  
A 2\((J + N)\)-dimensional vector for which the first \((J + N)\) components are the indices of basic variables in some order, and the next \((J + N)\) components are the indices of the complementary nonbasic variables in the same order. (Indexing of the variables follows the order in the vector \( z^t = (x^t, u^t, v^t, s^t) \).)

**JV**  
JV\((K)\) is the column location in \( \mathcal{W} \) of the variable indexed by \( K \).

**IR**  
IR\((K)\) is the row in \( \mathcal{W} \) labeled by the basic variable indexed by IB\((K)\), \( K \leq J + N \).

**B**  
This is the matrix \( D \) of the constraint set as used in the text.
Exhibit C-6

Linear Programming Under Uncertainty

Organization Chart of the Subroutines

SLPEXC → SLPROG
  ↓
  INSLP (— RANDM)
  ↓
  FILLW → NEWY(0)
  ↓
  INVRB → PVTSLP
  ↓
  TABLOW → PVTSLP
    ↓
    CHGIDX
    ↓
    STOREY
  ↓
  ITERAT → NULAST → NEWY(1)
  ↓
  STOREY
  ↓
  OUTSLP
### Exhibit C-7

**Linear Programming Under Uncertainty**

**Description of the Subroutines**

<table>
<thead>
<tr>
<th>Name of the Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLPexec</td>
<td>This is an executive subroutine which reads in the probability tables and then calls the main subroutine into operation to solve successive problems until no more are found on the input tape, whereupon an automatic terminal exit occurs.</td>
</tr>
<tr>
<td>SLPprog</td>
<td>This is a main subroutine which reads the identification card, checks the parameters of the problem, and then calls the functional subroutines as needed.</td>
</tr>
<tr>
<td>INSLP</td>
<td>This subroutine reads the input data, stores it, and also writes it on the output tape. (A special INSLP routine has been written for random data generation, and it is given last in Exhibit C-12.)</td>
</tr>
<tr>
<td>FILLw</td>
<td>For each major reapproximation, this subroutine fills the new tableau, calling upon NEwY to obtain the values of functions required; cf. Step 2, p. 44.</td>
</tr>
<tr>
<td>INVFB</td>
<td>This subroutine obtains the basic representation of the tableau, calling upon PVTSLP for the pivoting operation; cf. Step 3, p. 44.</td>
</tr>
<tr>
<td>TABLOW</td>
<td>This subroutine applies the Simplicial Algorithm on the basic representation of the tableau in order to iterate to an optimum for the approximating quadratic problem. It calls upon PVTSLP for the pivoting operation and CHGIDX for relabeling of the rows and columns of the tableau. At the end it calls STOREY to store the optimal y-values for use in the approximating procedure; cf. Step 4, p. 44.</td>
</tr>
<tr>
<td>ITERAT</td>
<td>This subroutine performs minor reapproximations by calling on NULAST and STOREY until an exit to OUTSLP, or back to TABLOW, is necessary; cf. Steps 6-8, p. 44.</td>
</tr>
<tr>
<td>NULAST</td>
<td>This subroutine calculates new values of the</td>
</tr>
</tbody>
</table>
Exhibit C-7 continued

column of basic variables for each minor reapproximation, obtaining the values of the functions involved by calling NEWY.

NEWY
This subroutine calculates function values as required for a major (KEY=0) or a minor (KEY=1) reapproximation. Currently it is written to handle only proportional losses and Normal probability distributions.

STOREY
This subroutine stores the solution values of the y-variables.

CHGIDX
This subroutine relabels pivot row and column of the tableau.

PUTSLP
This subroutine performs the pivoting operation on the tableau.

OUTSLP
This subroutine writes the solution values and a summary of the computations on the output tape.

Format of the Tableau

The tableau, which is called \( W \) in the routine instead of \( R^k \) and \( S^k \) as in the text, differs from (5-5) in that the columns of \( s^h \) and \( u^h \), and also \( s^c \) and \( u^c \), are interchanged. The vectors IB, JV, and IR and the matrix B are defined in Exhibit C-5. In this routine, the vector \( p \) of the text is called \( PI \), and the vector \( -q \) of the text is called \( P \); also, \( \text{GAMMA}(I) \) is the sum of \( \alpha_i \) and \( \beta_i \).
Variable-Factor Programming

Organization Chart of the Subroutines

BLEXEC → BLEND

BINPUT → BINOUT → BDATA → DATA (DORFMN)
  (BRANDM → BINOUT)

→ BPILLIT → BDATA → DATA (DORFMN)

BINVER → BPIVOT

BSOLVE → BPIVOT

BCONVR → BDATA → DATA (DORFMN)

BOUT
### Variable-Factor Programming

#### Description of the Subroutines

<table>
<thead>
<tr>
<th>Name of the Subroutine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEXEC</td>
<td>This is an executive subroutine which calls the main subroutine into operation to solve successive problems until no more problems are found and an automatic terminal exit occurs.</td>
</tr>
<tr>
<td>BLEND</td>
<td>This is the main subroutine, which calls the functional subroutines as needed.</td>
</tr>
<tr>
<td>BINPUT</td>
<td>This subroutine reads the ID card, checks the parameters, reads the input data, and calls BINPUT to write it out again on the output tape.</td>
</tr>
<tr>
<td>BINOUT</td>
<td>This subroutine writes the input data on the output tape, and it calls BDATA for writing of the function parameters.</td>
</tr>
<tr>
<td>BRANDM</td>
<td>This subroutine generates random input data when called by BINPUT.</td>
</tr>
<tr>
<td>BPILLT</td>
<td>For each major reapproximation this subroutine fills the new tableau, calling on BDATA for function values; cf., Step 2, p. 44.</td>
</tr>
<tr>
<td>BSOLVE</td>
<td>This subroutine performs the Simplicial Algorithm on the basic representation of the tableau to find an optimum for the approximating quadratic problem; cf., Step 4, p. 44.</td>
</tr>
<tr>
<td>BPIVOT</td>
<td>This subroutine performs the pivoting operation, and it also interchanges the labeling of the pivot row and column when called by BSOLVE; cf., pp. 30-31.</td>
</tr>
<tr>
<td>BCONVR</td>
<td>This subroutine performs minor reapproximations until either convergence is obtained, or a basic variable becomes negative; cf., Steps 6-8, p. 44.</td>
</tr>
<tr>
<td>BOUT</td>
<td>This subroutine writes out the solution values and a summary of the computations.</td>
</tr>
<tr>
<td>BDATA, DATA</td>
<td>These subroutines read in the yield-function parameters, and calculate values needed by the</td>
</tr>
</tbody>
</table>
approximation procedure. They call on DORFBN to obtain values for the agricultural planning problem mentioned in Section 6, and SULAPR to obtain values for the other functions listed at the end of Section 6. In practice, the user will write similar subroutines to calculate $\nabla \tilde{Y}$ and $\nabla^2 \tilde{Y}$. 
In order to distinguish this procedure from the Simplicial Algorithm described in Section 2, we shall call it the **Extended Simplicial Algorithm**:

1. Select an initial basis and an initial guess \((x^*; u^*)\).

2. Major reapproximation procedure: calculate \(\hat{w}_R^1\) from (4-3).

3. Obtain \(\hat{w}_R^1\) as in Step 1 of Figure (3-1).

4. Given \(\hat{w}_R^k\), use the Simplicial Algorithm for quadratic programming to obtain the solution to the approximating problem, and let \((x^{k+1}; u^{k+1})\) represent the solution values of the primal and dual program variables.

5. If the desired number of major reapproximations has not been made, return to Step 2 with \((x^*; u^*) = (x^{k+1}; u^{k+1})\).

6. Minor reapproximation procedure: obtain \(\hat{w}_L^{k+1}\) using (4-7).

7. Stopping procedure: if \(\left| \hat{w}_L^k - \hat{w}_L^{k+1} \right|\) is sufficiently small, so that a predetermined accuracy requirement has been met, then terminate the algorithm.

8. If \(\hat{w}_R^{k+1} \geq 0\) then return to Step 6; otherwise, return to Step 4 with \(\hat{w}_R^{k+1}\).

Except for Step 8, all of the steps in this algorithm are straightforward or familiar.

In Step 8 we encounter a new situation: we might return to Step 4 with a basic primal variable that has become negative in the course of the approximation procedure in Step 6, whereas the Simplicial Algorithm was not constructed to handle primal infeasibility. Remedies for this situation must, therefore, be our next topic.
SUBROUTINE S$OLER
(General Purpose Routine for Mathematical Programming)

BULLETIN NO. 2

USING S$OLER FOR QUADRATIC PROGRAMMING

August 3, 1964

S$OLER is designed to be a general purpose routine for mathematical programming, and among its abilities is quadratic programming. Due to the design of the optimization method employed in S$OLER, moreover, it takes complete advantage of the simplifying features of quadratic programming problems. In fact, in the special case of quadratic programming the method used in S$OLER consists only of Dentzig's Simplex Method.

Accompanying this Bulletin are a listing and a card deck of routines to be used with S$OLER for quadratic programming. There is a MAIN routine, here titled "MAINQP", which accomplishes the input and output operations, and which calls S$OLER for the optimization procedures. In addition, there is a VALUE function, here titled "VALQQP", which is called by S$OLER for the functional values required in the optimization procedures. These routines are written in the FORTRAN II language, but FORTRAN IV versions are available on request; also, one can use IEM's "SIFT" routine to convert the FORTRAN II deck into a FORTRAN IV deck (SIFT is available at most installations using FORTRAN IV).

To use these routines one prepares a package of the FORTRAN source decks and a data deck of the following form:

* FORTRAN
  [deck MAINQP]
* FORTRAN
  [deck VALQQP]
* FORTRAN
  [deck S$OLER]
* DATA
  [data deck]
The composition of the data deck is described below.

The quadratic programming problem can be formulated as follows:

Maximize \( F_0(X) = P'X + \frac{1}{2}X'QX \)

Subject to

1. a) \( X_j \) unrestricted if \( IQ(J) = 0 \),
   b) \( X_j \geq 0.0 \) if \( IQ(J) \neq 0 \), for \( 1 \leq J \leq NV \); and if \( NC > 0 \),

2. a) \( F_K(X) = 0.0 \) if \( IQ(NV+K) = 0 \),
   b) \( F_K(X) \geq 0.0 \) if \( IQ(NV+K) \neq 0 \), for \( 1 \leq K \leq NC \),

where, \( F(X) = (F_K(X)) = B - AX \).

In this formulation, \( NV \) is the number of variables, and \( NC \) is the number of constraints; \( IQ \) is a vector specifying which inequality constraints are imposed, if any. The matrix \( Q \) must be negative semi-definite in order for \( F_0(X) \) to be concave. The dimensions of the vectors and matrices are as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>No. of Rows</th>
<th>No. of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>NV</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>NV</td>
<td>1</td>
</tr>
<tr>
<td>Q</td>
<td>NV</td>
<td>NV</td>
</tr>
<tr>
<td>B</td>
<td>NC</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>NC</td>
<td>NV</td>
</tr>
<tr>
<td>IQ</td>
<td>NV+NC</td>
<td>1</td>
</tr>
<tr>
<td>IBB</td>
<td>NV+NC</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>NC</td>
<td>1</td>
</tr>
<tr>
<td>V</td>
<td>NV</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>NC</td>
<td>1</td>
</tr>
</tbody>
</table>

Besides the items already mentioned this table includes \( IBB, U, V, \) and \( S \). \( U_K \) is the dual variable for constraint \( K \) as in (2) above; \( V_j \) is the dual variable for the restriction on \( X_j \) as in (1) above; \( S_K = F_K(X) \) is the slack in constraint \( K \); and
3. a) for \(1 \leq J \leq NV\),
\[
IBB(J) \neq 0 \text{ if } X_J \text{ is basic,}
\]
\[
IBB(J) = 0 \text{ if } X_J \text{ is nonbasic,}
\]
b) for \(1 \leq K \leq NC\),
\[
IBB(NV+K) \neq 0 \text{ if } U_K \text{ is basic,}
\]
\[
IBB(NV+K) = 0 \text{ if } U_K \text{ is nonbasic.}
\]
Note that \(U_K\) is basic if and only if \(F_K(X) = 0.0\) by constraint of either type (2a) or (2b).

The data deck amounts to a sequence of item packages each consisting of a format card (except for IQ and IBB) followed by the data values described by the format card*. The sequence is as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Format card NV, NC</td>
<td>2.3 (\Delta)</td>
</tr>
<tr>
<td>2</td>
<td>Format card ((P(J),J = 1,NV))</td>
<td>6.0 (\Delta)</td>
</tr>
<tr>
<td>3</td>
<td>Format card (((Q(I,J),I = 1,NV),J = 1,NV))</td>
<td>-1.0, 0.1, 0.1-1.0 (\Delta)</td>
</tr>
<tr>
<td>4</td>
<td>Format card ((B(K),K = 1,NC))</td>
<td>5.0-2.0-2.0 (\Delta)</td>
</tr>
<tr>
<td>5</td>
<td>Format card (((A(K,J),J = 1,NV),K = 1,NC))</td>
<td>1.0, 0.1-1.0-0.2-0.2-1.0 (\Delta)</td>
</tr>
<tr>
<td>6</td>
<td>((IQ(N),N = 1, NV+NC))</td>
<td>11111</td>
</tr>
<tr>
<td>7</td>
<td>((IBB(N),N = 1, NV+NC))</td>
<td>00000</td>
</tr>
</tbody>
</table>

Note that the formats of IQ and IBB are each fixed as a sequence of NV+NC \(\leq 70\) integer fields of length one (1). For IBB one should specify his initial best guess of the optimal basis: this initial basis need not be primal feasible but it must be consistent (number of nonbasic X's plus number of basic U's \(\leq NV\)).

Although the matrix \(Q\) is here read in its entirety, \(\text{SOLVER}\) calls for

* For instructions on how to prepare a format card, see any one of the IBM manuals describing FORTRAN II or IV; e.g., IBM Form C28-6054-3.
Q(I,J) only for $J \leq I$.

A sequence of quadratic programming problems can be solved by placing their respective data decks in sequence.

MAINFQP writes out the input it has read in, except for IQ and IEB, and after the solution has been obtained by S\$LVER it writes out the optimal $X,U,V,S$ vectors. In addition, it writes out the value of a variable "KAPUT" on the far left-hand-side of the page. This value is to be interpreted as follows:

<table>
<thead>
<tr>
<th>KAPUT</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Solution obtained without difficulty</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>No solution obtained</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Solution obtained with difficulty (e.g., S$LVER may have detected that $Q$ is not negative-semi-definite, in which case the solution obtained may perhaps be only a local solution)</td>
</tr>
</tbody>
</table>

For a description of the further significance for debugging purposes of the nonzero values of KAPUT, see the S\$LVER DESCRIPTION and GLOSSARY.

As currently set up the dimension statements in the routines permit the following maximum parameter values: $NV \leq 50$, $NC \leq 50$, $NV + NC \leq 70$. Judicious consideration of the memory allocation, however, can enable one to extend these limits.
### SUBROUTINE SOLVER

#### GLOSSARY

<table>
<thead>
<tr>
<th>FORTRAN</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. External Functions</strong></td>
<td></td>
</tr>
</tbody>
</table>
| VALUE (X,I,J,K)  | \[ \frac{\partial^2 F_K(X)}{\partial X_I \partial X_J}, \text{ all } K, \text{ if } 0 < J \leq I; \]
|                  | \[ \frac{\partial F_K(X)}{\partial X_J}, \text{ all } K, \text{ if } 0 = I < J; \]
|                  | \[ F_K(X), \text{ if } K \neq 0, \text{ for } 0 = I = J. \]
<p>|                  | The user supplies VALUE.                                                                                                                  |
| <strong>2. External Vectors (Fixed-point)</strong> |                                                                                                                                           |
| IBB(N)           | IBB(N) ( \neq 0 ) if and only if VARIABLE(N) is basic, where VARIABLE(N) = X(N) if N ( \leq ) NV, and VARIABLE(N) = U(N-NV) if NV &lt; N ( \leq ) NV + NC. |
|                  | Initially, the user supplies his guess of the optimal basic. Note: IBB is used for temporary storage during the basis inversion; the optimal basis is returned at the end. |
| IQ(N)            | IQ(N) ( \neq 0 ) if and only if CONSTRAINT (N) ( \geq 0 ) is required for a solution, where CONSTRAINT(N) = X(N) if N ( \leq ) NV, and CONSTRAINT(N) = [ \frac{F}{N-NV}(X) ] if NV &lt; N ( \leq ) NV + NC. |
|                  | The user supplies IQ.                                                                                                                     |</p>
<table>
<thead>
<tr>
<th>FORTRAN</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. External Vectors (Floating-point)</td>
<td></td>
</tr>
<tr>
<td>S(K)</td>
<td>$S(K) = F_k(X)$, the slack in constraint $K$.</td>
</tr>
<tr>
<td>U(K)</td>
<td>U(K) is the dual (price) variable associated with constraint $K$; initially, the user provides his own best guess about the optimal value of $U$.</td>
</tr>
<tr>
<td>V(J)</td>
<td>V(J) is the dual (price) variable associated with the constraint $X(J) \geq 0$; if $X(J)$ is not constrained to be nonnegative, then $V(J) = 0$.</td>
</tr>
<tr>
<td>X(J)</td>
<td>X(J) is a primal program, or decision, variable; initially, the user supplies his own estimate of the optimal value of $X$. Note: $X$ is an explicit argument of the external function $VALUE$.</td>
</tr>
<tr>
<td>4. External Variables</td>
<td></td>
</tr>
<tr>
<td>EPSILN</td>
<td>Maximum percentage error which is tolerable in the solution values for $X(J)$ and $U(K)$. Initially this value is supplied by the user; at the end, the actual maximum percentage error in the solution values is returned as $EPSILN$.</td>
</tr>
<tr>
<td>ITMAX</td>
<td>Maximum number of complete quadratic approximations allowable to obtain convergence within</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>DEFINITION</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>the existing limitations of computer time.</td>
<td></td>
</tr>
<tr>
<td>Initially this value is supplied by the user;</td>
<td></td>
</tr>
<tr>
<td>at the end, the actual number of quadratic</td>
<td></td>
</tr>
<tr>
<td>fittings required to obtain the solution is</td>
<td></td>
</tr>
<tr>
<td>returned as ITMAX.</td>
<td></td>
</tr>
<tr>
<td>Note: set ITMAX = 0 for linear and quadratic</td>
<td></td>
</tr>
<tr>
<td>programming problems.</td>
<td></td>
</tr>
<tr>
<td>At the end, if</td>
<td></td>
</tr>
<tr>
<td>a) KAPUT = 0, then a solution was obtained,</td>
<td></td>
</tr>
<tr>
<td>without difficulty;</td>
<td></td>
</tr>
<tr>
<td>b) KAPUT &gt; 0, then a solution was obtained,</td>
<td></td>
</tr>
<tr>
<td>but trouble was encountered along the</td>
<td></td>
</tr>
<tr>
<td>way; in this case KERRØR = -1-KAPUT, an</td>
<td></td>
</tr>
<tr>
<td>internal variable (see below), and it can</td>
<td></td>
</tr>
<tr>
<td>be factored into primes to identify the</td>
<td></td>
</tr>
<tr>
<td>sources of trouble;</td>
<td></td>
</tr>
<tr>
<td>c) KAPUT &lt; 0, then no solution was obtained;</td>
<td></td>
</tr>
<tr>
<td>here, KERRØR = KAPUT, and it can be fac-</td>
<td></td>
</tr>
<tr>
<td>tored to identify the sources of trouble.</td>
<td></td>
</tr>
<tr>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>NC is the number of constraints,</td>
<td></td>
</tr>
<tr>
<td>0 ≤ NC &lt; 70, NC = dimension (S) = dimension (U).</td>
<td></td>
</tr>
<tr>
<td>NV</td>
<td></td>
</tr>
<tr>
<td>NV is the number of primal program variables,</td>
<td></td>
</tr>
<tr>
<td>1 ≤ NV ≤ 70, NV = dimension (X) = dimension (V).</td>
<td></td>
</tr>
<tr>
<td>Note: NV + NC ≤ 70.</td>
<td></td>
</tr>
<tr>
<td>FORTRAN</td>
<td>DEFINITION</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5. Internal Matrices</td>
<td>R(I,J) for J &lt; LAST is the tableau for the quadratic programming procedures, column</td>
</tr>
<tr>
<td></td>
<td>LAST contains the solution values for the quadratic program, and columns J = LAST + 1,</td>
</tr>
<tr>
<td></td>
<td>..., LASTN contain the inverse of the matrix of basic columns.</td>
</tr>
<tr>
<td>6. Internal Vectors (Fixed-point)</td>
<td>If IB(N) = L then VARIABLE(L) is basic if and only if N ≤ NV + NC = NN*. Note: if N ≤ NN, then IB(N + NN) is the primal or dual variable complementary to L, unless there is a floating pivot outstanding.</td>
</tr>
<tr>
<td>IB(N)</td>
<td>IR(N) is the row location in the tableau of the (basic) variable IB(N) occupying location N of the basis description.</td>
</tr>
<tr>
<td>IR(N)</td>
<td>JC(L) is the location in the tableau of the column corresponding to variable L*.</td>
</tr>
<tr>
<td>JC(L)</td>
<td>*Note: VARIABLE(L) = X(L) if L ≤ NV;</td>
</tr>
<tr>
<td></td>
<td>= U(L-NV) if NV &lt; L ≤ NV + NC;</td>
</tr>
<tr>
<td></td>
<td>= V(L-NV-NC) if NV + NC &lt; L ≤ 2*NV + NC;</td>
</tr>
<tr>
<td></td>
<td>= S(L-2<em>NV-NC) if 2</em>NV + NC &lt; L ≤ 2*(NV + NC)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>DEFINITION</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7. Internal Vectors</td>
<td><strong>XSTAR(N)</strong> is defined as follows for N ≤ NV:&lt;br&gt; XSTAR(N) = X(N) if N ≤ NV&lt;br&gt; XSTAR(N) = U(N-NV) if NV &lt; N ≤ NV + NC&lt;br&gt;</td>
</tr>
<tr>
<td>(Floating-point)</td>
<td>for the X and U values at which a complete quadratic approximation was last made.</td>
</tr>
<tr>
<td>8. Internal Variables</td>
<td><strong>EPZERØ</strong> is the smallest positive value to be considered different from zero; e.g., in determining singularity of a matrix; also, in deciding whether a constraint is satisfied.</td>
</tr>
<tr>
<td>(Floating-point)</td>
<td>Temporary storage.</td>
</tr>
<tr>
<td>A, AA, AD, AP, DEL</td>
<td></td>
</tr>
<tr>
<td>(Fixed-point)</td>
<td><strong>IBFP</strong> The floating pivot variable.*</td>
</tr>
<tr>
<td></td>
<td><strong>IP</strong> The pivot row in the tableau.</td>
</tr>
<tr>
<td></td>
<td><strong>IPN</strong> IB(IPN) is the pivot variable;* therefore, IPN is the location of the pivot variable in the basis description.</td>
</tr>
<tr>
<td></td>
<td><strong>IPPN</strong> IB(IPPN) is the primal or dual complementary variable to the pivot variable;* therefore, IPPN = NN + IPN is the location of the complementary variable in the basis description.</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>DEFINITION</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>IRFP</td>
<td>IRFP = IR(NFP) is the row location in the tableau of the floating pivot variable.*</td>
</tr>
<tr>
<td>JP</td>
<td>The pivot column in the tableau.</td>
</tr>
<tr>
<td>KERRØR</td>
<td>Each time trouble occurs, KERRØR is multiplied by a prime number as follows:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prime</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>No convergence, KQFIT = ITMAX;</td>
</tr>
<tr>
<td>3</td>
<td>Singular matrix of basic columns encountered;</td>
</tr>
<tr>
<td>5</td>
<td>Problem appears to lack concavity in some of the functions involved;</td>
</tr>
<tr>
<td>7</td>
<td>Unbounded solution indicated.</td>
</tr>
<tr>
<td>KLFIT</td>
<td>Number of first-order (linear) improvements since the last complete quadratic approximation to the saddle function.</td>
</tr>
<tr>
<td>KP</td>
<td>KP = IB(IPN) is the pivot variable.*</td>
</tr>
<tr>
<td>Note:</td>
<td>KP is used for temporary storage during the linear fitting procedure.</td>
</tr>
<tr>
<td>KPØWER</td>
<td>Cf. statement no. 93. KPØWER is the exponent applied to KQFIT to determine the allowable maximum for KLFIT.</td>
</tr>
<tr>
<td>KQFIT</td>
<td>Number of complete quadratic approximations to the saddle function executed so far; KQFIT ≤ ITMAX; the final KQFIT value is returned as ITMAX.</td>
</tr>
</tbody>
</table>

* See footnote on page 4.
<table>
<thead>
<tr>
<th>FORTRAN</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAST</td>
<td>LASTN = LAST + NN is the number of columns in the tableau; LASTN ≤ 211.</td>
</tr>
<tr>
<td>NFD</td>
<td>Temporary storage for location in basis description of dual floating pivot variable.</td>
</tr>
<tr>
<td>NFP</td>
<td>Location in basis description of the floating pivot variable.</td>
</tr>
<tr>
<td>NN</td>
<td>NN = NV + NC = dimension (X) + dimension (U), the sum of the numbers of variables and constraints in the problem; NN ≤ 70.</td>
</tr>
<tr>
<td>NP</td>
<td>The location in the basis description of the first nonbasic variable.</td>
</tr>
<tr>
<td>NPC</td>
<td>Location in the basis description of the nonbasic variable to be made basic.</td>
</tr>
<tr>
<td>NQ</td>
<td>The number of inequality constraints ((X(J) \geq 0 \text{ or } F_K(X) \geq 0)) in the problem; (0 \leq NQ \leq NN).</td>
</tr>
<tr>
<td>NVP</td>
<td>NVP = NV + 1.</td>
</tr>
</tbody>
</table>

10. Internal Switching Variables

| KPIVØT  | Exit from pivoting operation. |
| KVARID  | Exit from identification of variable values. |
SUBROUTINE SOLVER*

INSTRUCTIONS

SOLVER is a FORTRAN** subroutine which solves mathematical programming problems of the following type:

Maximize  \( F_0(x) \)

Subject to

1. a) \( x(j) \) unrestricted if \( IQ(j) = 0 \),
   b) \( x(j) \geq 0.0 \) if \( IQ(j) \neq 0 \),
for \( 1 \leq j \leq NV \), and
2. a) \( F_k(x) = 0.0 \) if \( IQ(NV + k) = 0 \),
   b) \( F_k(x) \geq 0.0 \) if \( IQ(NV + k) \neq 0 \),
for \( 1 \leq k \leq NC \),

where \( 1 \leq NV \leq 70, \ 0 \leq NC < 70, \ NV + NC \leq 70 \),
and each function \( F_k(x) \), \( 0 \leq k \leq NC \), is concave and twice differentiable (at least within a neighborhood of the solution).

In this formulation, \( NV \) is the dimension of the vector \( x \), and \( NC \) is the number of constraints (which can be zero). \( IQ \) is a vector specifying which inequality constraints are to be imposed, if any.

SOLVER can be called by a main routine of the following form:

DIMENSION \( X(70), U(70), V(70), S(70), IBB(70), IQ(70) \)

\( F \)  VALUE

CALL INPUT (\( X, U, IBB, IQ, NV, NC, ITMAX \))

\( EPSILN = 1.0E - 03 \)

* May, 1963; revised May, 1964

**SOLVER can be compiled by IBM's FORTRAN II and FORTRAN IV compilers for the 7090/7094 machines, and by CDC's FORTRAN-compatible compiler for the 1604.
CALL SOLVER (X, U, V, S, IBB, IQ, NV, NC, ITMAX, EPSILN, KAPUT, VALUE)
IF (KAPUT) 1, 2, 2
1 CALL ERROR
2 CALL OUTPUT (X, U, V, S, IBB, ITMAX, EPSILN, KAPUT)
   .L占据 EXIT
END

The additional features of this calling sequence are explained below.

1. The DIMENSION of every one of the vectors must be 70, unless SOLVER's internal DIMENSION statements are altered appropriately. The latter may be necessary for large problems, or if the function VALUE consumes a large part of the available core memory. With NV + NC ≤ 70, SOLVER requires about 17300 words of memory for its own use.

2. The function VALUE is supplied by the user. Its name here is a dummy name, and actually may have any name desired. If a FORTRAN IV compiler is to be used, the "F VALUE" card is replaced by an "EXTERNAL VALUE" card.*

VALUE is defined by a FORTRAN subprogram:

FUNCTION VALUE (X, I, J, K)

DIMENSION X(70)
   ..
   ..
   ..
RETURN
END

which returns values as follows:

\[
VALUE (X, I, J, K) = \begin{cases} 
F_K(X) & \text{if } 1 \leq K \leq NC, J = I = 0; \\
3 F_K(X)/2X_J & \text{if } 0 \leq K \leq NC, J > I = 0; \\
9 F_K(X)/2X_J^2 & \text{if } 0 \leq K \leq NC, I \geq J > 0.
\end{cases}
\]

Note: if NC = 0 then \(VALUE (X, I, J, K) = 0.0\) for \(K > 0\).

*In FORTRAN IV it is possible to modify SOLVER so as to employ variable DIMENSION specifications.
3. ITMAX specifies the maximum number of successive approximations allowed to obtain the solution. For linear or quadratic programming, ITMAX = 0; otherwise, a value of ITMAX in the range \([3, 10]\) is reasonable (but not required), and general purpose values are ITMAX = 4 for small problems and ITMAX = 7 for larger problems. At the end, the actual number of approximations required is returned as ITMAX.

4. EPSILN is the maximum relative error allowed in the solution values of \(X(J)\) and \(U(K)\); e.g., EPSILN = 1.0'0^3 allows a 0.1\% error. At the end, the actual relative error bound is returned as EPSILN.

5. Initially, the user supplies his own best guesses for the solution values of the vectors \(X, U,\) and \(IBB,\) where (a) \(X\) is the vector of primal program variables to be optimized, (b) \(U\) is the vector of dual program variables (viz., \(U(K), 1 \leq K \leq NC,\) is the dual variable for the \(K^{th}\) constraint), and (c) \(IBB\) is the basis description in the following form:

   (i) \(IBB(J) = 0\) if and only if \(X(J)\) is constrained to zero;

   (ii) \(IBB(NV+K) = 0\) if and only if \(U(K)\) is constrained to zero.

The initial specification of \(IBB\) need not be a feasible basis, but it must be consistent. Note that \(U(K)\) is constrained to zero if and only if the constraint \(F^*_K(X) \geq 0.0\) is not binding. At the end, SOLVER returns the optimal values of \(X, U, V, S,\) and \(IBB,\) where (a) \(V(J), 1 \leq J \leq NV,\) is the dual variable for the constraint \(X(J) \geq 0.0,\) or is zero if \(X(J)\) is unrestricted, and (b) \(S(K), 1 \leq K \leq NC,\) is the slack in the \(K^{th}\) constraint, \(S(K) = F^*_K(X) \geq 0.0\).

6. KAPUT is an error signal for the user. If KAPUT = 0 then the solution was obtained without difficulty. If KAPUT > 0 then the solution was obtained, but certain potential difficulties were encountered in the process. If KAPUT < 0 then no solution was obtained. SOLVER calculates KAPUT as the
product of certain prime numbers, so that by factoring an error signal value one can trace the source of trouble; for a description of the prime number references, the user is referred to the subroutine listing and to the SOLVER Glossary, pages 3 and 6, which has been published separately.

7. The subroutines INPUT, ERRR, and OUTPUT are, of course, supplied by the user.

Further references on SOLVER are the SOLVER Description and the SOLVER Glossary; also,

1. Introduction

SOLVER is a computer routine which solves mathematical programming problems. Written as a subroutine in the FORTRAN language, its main strengths are (1) that it is applicable to virtually the entire range of practical nonlinear programming problems, and yet (2) that it is quick and simple to use. In addition, because the FORTRAN language is easily understood, and because the computational method employed in the routine is straightforward, SOLVER is open to scrutiny, modification, and improvement. For the same reasons, SOLVER is especially appropriate for instructional purposes. Another feature of some practical importance is that SOLVER enables one to make complete use of prior information about the optimal solution to a problem.

Chief among SOLVER's limitations are that (1) since it is written in a macrolanguage, it lacks the ultimate degree in computer (as versus man) time-saving efficiency, (2) as written it handles a total of only 70 variables and constraints, and (assuming a 32,000 word memory in the machine) modification can not extend this number beyond 100 without resorting to tape storage, and (3) it requires that the objective function and each of the constraint functions be concave and twice differentiable within a neighborhood of the solution. Local concavity will suffice if one specifies the locality in the prior guess of the solution. Because it is designed for general-purpose use, of course, SOLVER does not exploit simplifying structural features sometimes present in special problems.

2. This short description is not intended to be complete. Preliminary analyses were presented in [6] and [7], and an extensive exposition was given in [5]. A complete description will be published as a Working Paper of the Western Management Science Institute, and will be submitted for publication in a technical journal.
3. Actually by artificially "smoothing" discontinuous second derivatives, with
SOLVER handles linear and nonlinear objective functions and constraint functions with equal ease, and it takes complete advantage of the simplifying features of quadratic programming problems. It is, however, time-consuming for linear programming compared to the specialized routines available. SOLVER accepts problems either with or without constraints, and any mixture of equality and inequality constraints is allowable, as well as any mixture of free and sign-restricted variables. It can even be used for solving ordinary simultaneous equations, linear or nonlinear.

2. Formulation

Briefly stated, SOLVER's job is to solve concave programming problems, of the following type. Let the (NC+1) functions \( \{F_K(X) ; K = 0, \ldots, NC\} \) be concave and twice-differentiable functions of the NV-dimensional vector \( X \).

\[
\text{Maximize } F_0(X) \\
\text{Subject to}
\]

1. a) \( X(J) \) unrestricted if \( I^c(J) = 0 \),
   
b) \( X(J) \geq 0.0 \) if \( I^c(J) \neq 0 \),
   
   for \( 1 \leq J \leq NV \), and if \( NC > 0 \),

2. a) \( F_K(X) = 0.0 \) if \( I^c(NV+K) = 0 \),
   
b) \( F_K(X) \geq 0.0 \) if \( I^c(NV+K) \neq 0 \),
   
   for \( 1 \leq K \leq NC \).

In this formulation, NV is the number of variables (viz., the dimension of the vector \( X \)), and NC is the number of constraints (which can be zero). \( I^c \) is an

3. (Contd.) Some skill one can handle a more general class of problems. Discontinuous first derivatives, however, are likely to cause trouble if the solution occurs at a breakpoint.

* We will use the notations \( X(J) \) and \( X_j \) interchangeably to denote vector components.
(NV+NC)-dimensional vector specifying which inequality constraints are to be imposed, if any.

In order to simplify the discussion in what follows, we will consider all variables to be sign-restricted as in 1b, and all constraints to be inequalities as in 2b. The generalization to unrestricted variables and equality constraints is straightforward, and can be omitted until the end.

3. Conditions for a Solution

The fundamental result in mathematical programming was obtained by H. Kuhn and A. Tucker [4]. Specifically, regardless of the concavity and differentiability assumptions it is sufficient, and with the concavity assumption plus a mild constraint qualification it is necessary, for \( X^0 \) to be a solution to the Programming Problem, that there exist a solution \( (X^0, U^0) \) for \( \delta(X,U) = F_0(X) + \sum_{K=1}^{NC} U_K F_K(X) \) to the following

\[
\text{SADDLE-VALUE PROBLEM} \tag{2}
\]

Find vectors \( (X^0, U^0) \) such that

\[
\delta(X,U^0) \leq \delta(X^0,U^0) \leq \delta(X^0,U)
\]

for all \( X \geq 0, U \geq 0 \).

Here, \( X_J \) is unrestricted if 1a obtains in the Programming Problem, and \( U_K \) is unrestricted if 2a obtains. The Kuhn-Tucker result states in effect that we can direct our attention towards solving the Saddle-Value Problem, which can be simply characterized in the format

\[
\begin{align*}
\text{Maximize} & \quad \left[ F_0(X) + \sum_{K=1}^{NC} U_K F_K(X) \right] \\
\text{subject to} & \quad X \geq 0, \quad U \geq 0
\end{align*}
\tag{3}
\]

The conditions for a solution to the Saddle-Value Problem are easily derived, and can be succintly stated in the following
SUMMARY OF THE CONDITIONS FOR A SOLUTION

A. Find nonnegative Program Vectors Slack Vectors

<table>
<thead>
<tr>
<th>Primal Vectors</th>
<th>X</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Vectors</td>
<td>U</td>
<td>V</td>
</tr>
</tbody>
</table>

B. Solving the simultaneous equations

\[ \nabla F_0(X) + U' \nabla F(X) + V' = 0 \]
\[ \begin{align*}
-F(X) + S &= 0 \\
V'X &= 0.0 & U'S &= 0.0 & \text{are satisfied.}
\end{align*} \]

C. Such that the "complementary slackness" conditions

With these conditions in hand, the standard methodology for obtaining a solution (which we shall follow with only slight modification) is to search among candidates satisfying parts B and C to find a solution satisfying part A. When detailed refinements are included, this is Dantzig's Simplex Method, and modifications on this methodology are sometimes called simplicial methods.

4. Motivation for the Computational Method

For the special cases of linear and quadratic programming, the simultaneous equations in part B are linear, and therefore they present no analytical or computational difficulties. In these cases attention can be directed towards a search procedure for finding a solution satisfying part A, and for this purpose Dantzig's Simplex Method for quadratic programming [1]

---

4. The gradient operator \( \nabla \) applied to a column vector \( F(X) = (F_i(X)) \) yields the Jacobian matrix \( (\frac{\partial F_i(X)}{\partial x_j}) \) of first partial derivatives. Later, the iterated operator \( \nabla^2 \) will be used to specify the symmetric Hermitian matrix of second partial derivatives,

\[ \nabla^2 F_k(X) = \left( \frac{\partial^2 F_k(X)}{\partial x_i \partial x_j} \right) \]
is quite effective.

When the equations in part B are nonlinear, however, a substantive problem arises in the repeated need to solve them in the course of the search procedure. It is, in fact, computationally infeasible to solve them repeatedly for each and every candidate for the solution. SOLVER exploits one way out of this difficulty: it solves a succession of approximating quadratic programming problems (for which the equations of part B are linear), the solutions to which converge to the solutions of the actual Programming Problem within any desired degree of accuracy.

The rationale of this approach will be spelled out below in Section 5, along with the technical embellishments that make it computationally efficient. Here, we will develop an alternative formulation possessing more intuitive appeal and greater cogency: the algorithm which results, nevertheless, is exactly the same.

For the Programming Problem to be solved, the objective function in the Saddle-Value Problem is

\[ \hat{f}(x,u) = \Phi(x) + \sum_{k=1}^{nk} U_k F_k(x). \]  

(5)

We consider approximating \( \hat{f} \) by the objective function in the saddle-value problem for a quadratic programming problem; viz., for an approximating problem of the form,

Maximize \[ P'X + \frac{1}{2}X'QX \]  

Subject to \[ X \geq 0, \quad AX \leq B, \]

\( \hat{f} \) is to be approximated by

\[ \hat{x}(x,u) = P'X + \frac{1}{2}X'QX + U'[B - AX]. \]  

(7)

For this approximation we shall anticipate the possibility of technical refinements for greater computational efficiency by allowing for a succession of first-order (linear) fittings in-between complete second-order (quadratic)
fittings.

Second-order fitting conditions are imposed at a sequence of points to be denoted by \((X^*, U^*)\); and then following each second-order fitting, first-order fitting conditions are imposed at a sequence of points indexed by \(N\) and denoted by \((X^{(N)}, U^{(N)})\). For easier reference, the * and \(N\) designations will be carried over in our notation for the approximation parameters \((0, A; P, B)\). The fitting conditions are given below.

\[ \nabla^2 \hat{s}(X^*, U^*) = \nabla^2 \hat{s}(X^*, U^*) \quad (8) \]

\[
Q^* = \nabla^2 F_0(X^*) + \sum_{K=1}^{NC} U^*_K \nabla^2 F_K(X^*) \\
A^* = -\nabla F(X^*)
\]

**FIRST-ORDER FITTING CONDITIONS:**

\[
\nabla \hat{s}(X^{(N)}, U^{(N)}) = \nabla \hat{s}(X^{(N)}, U^{(N)}) \quad (9) \\
P^{(N)} + Q^* X^{(N)} - [A^*]' U^{(N)} = \nabla F_0(X^{(N)}) + [\nabla F(X^{(N)})]' U^{(N)} \\
B^{(N)} - A^* X^{(N)} = F(X^{(N)})
\]

Observe that the second-order conditions suffice to determine the matrices \(Q^*\) and \(A^*\), and then with these in hand we can determine the vectors \(P^{(N)}\) and \(B^{(N)}\) from the first-order conditions. This is a clue to the reason for interspersing the less effective first-order fittings in between the complete second-order fittings: although this refinement increases the number of fittings needed to obtain convergence, it has the advantage of reducing substantially the number of matrix inversions required, and thereby it minimizes the total computing time! It can be noted here parenthetically that in \(SOLVER\) each series of first-order fittings is initiated at \((X^{(1)}, U^{(1)}) = (X^*, U^*)\).
The parameters \((Q^*, A^*; p^{(N)}_1, b^{(N)}_1)\) determined by the fitting conditions provide the data for the solution of the approximating quadratic problem, which is accomplished in SOLVER by Dantzig's Simplex Method. After the solution to the approximating problem is obtained, a new approximation is made, and so on. In this fashion the process is repeated until convergence is attained. Of course, if the initial guess \((X^0, U^0)\) is too far from the optimum, then the possibility exists that this process will not converge, and this contingency will be examined in more detail in Section 6*: when this happens, SOLVER's only recourse is to ask you to take advantage of what has been learned and to try again with a better choice of the initial guess.

Figure 1 presents a flow chart of the computational sequence in SOLVER. In practice, SOLVER pursues convergence of the first-order fitting process only for a limited number of iterations, after which it initiates a new second-order fitting and continues from there.

One should note also that ordinarily SOLVER will identify the optimal basis for the Programming Problem very early in the process, since the optimal basis is insensitive to minor errors of approximation; consequently, thereafter the process in effect consists solely of successive approximations to the solution of the equations in part B of the Conditions for a Solution**.

5. The Algorithm

The formulation in this section will be more technical, and more general as well. It will be more technical in that the procedural steps of the

---

* It is worth remarking, however, that the concavity of the functions involved in the Programming Problem tends to contribute to an extensive neighborhood of convergence.

** With reference to the discussion in Section 5, the following characterization can be drawn. Whereas general simplicial methods solve the required simultaneous equations by Newton's Method for every candidate encountered in the search procedure, SOLVER reverses roles and for each iteration of Newton's Method optimizes the choice of basis by using the search procedure. One effect of this reversal is that the optimal basis is found quickly, since it is insensitive to minor errors of approximation.
Figure 1
Flow Chart of SOLVER
algorithm will be developed in detail. And it will be more general in that it will develop a general method which includes S\$LVER's algorithm for mathematical programming as a special case. The general method is an extension of Newton's Method [2,3] for finding roots of equations. When the equations take the form of those in part B of the Kuhn-Tucker Conditions for a Solution to the Programming Problem, then we have the special case of mathematical programming.

Consider the following general problem. Let $H(\cdot, \cdot)$ be an $L$-dimensional vector-valued function of two $L$-dimensional vectors, say $Y$ and $Z$. We shall assume that $H$ is differentiable over the relevant domain. The problem is to find roots of the homogeneous equation

$$H(Y^0, Z^0) = 0 \quad (10)$$

such that $Y^0 \geq 0$, $Z^0 \geq 0$, and $(Y^0)' Z^0 = 0.0$.

[The Programming Problem is the special case in which (10) is the set of simultaneous equations in part B of the Conditions for a Solution, with $Y' = (X', U')$ and $Z' = (V', S')$. That is, $L = NV + NC$, $Y$ is the vector of primal and dual program variables, and $Z$ is the vector of primal and dual slack variables.]

For expository convenience, let us adopt a uniform notation which is consistent with the structure of the problem. The "complementary slackness" conditions appended to (10) mean that (by appropriate reordering of the components) the vector $W' = (Y', Z')$ can be broken up into two parts: (1) the $L$-dimensional vector $W_N$ of nonbasic components which are restricted to zero, and (2) the complementary $L$-dimensional vector $W_B$ of basic components which are not restricted to zero. In this notation, the complementary slackness conditions are construed as $W_N^0 = 0$, $W_B^0 \geq 0$, and $Y_J^0$ is a component of $W_B^0$ if and only if $Z_J^0$ is a component of $W_N^0$ and vice versa. The equation (10) we shall write as

$$H(W_B^0, W_N^0) = 0 \text{ , or } H(W_B^0, W_N^0) = 0 \quad (11)$$
conditional on the choice of basis $B$.

Following the lead of Newton's Method, we shall solve our general problem by constructing a new function $G(W_B, W_N)$ such that $W_B^\circ$ and $W_N^\circ$ are obtained as the limit of an iterative sequence,

$$W_B^\circ = \lim_{N \to \infty} W_B^{(N)} ,$$

(12)

$$W_B^{(N)} = G(W_B^{(N-1)}, W_N^{(N-1)}) ,$$

(13)

and

$$W_N^{(N)} = 0 .$$

(14)

The function $G$ can be constructed in a variety of ways, and in fact many different forms are used in modern numerical analysis [2]. In the form with which we are concerned, namely Newton's Method, $G$ is derived by taking a tangential approximation to $H$ at a point $W^*$.

Approximate $H$ by its tangent hyperplane at $W^*$ as follows. Let $\nabla_B$ and $\nabla_N$ be the gradient operators restricted to the basic and nonbasic variables, respectively. Define

$$\tau_B^* = \nabla_B H(W_B^*, W_N^*)$$

and

$$\tau_N^* = \nabla_N H(W_B^*, W_N^*) .$$

(15)

Then the approximation is given by

$$H(W_B, W_N) \approx H(W_B^*, W_N^*) + [\tau_B^*, \tau_N^*] \left\{ \begin{bmatrix} W_B^* \\ W_N^* \end{bmatrix} - \begin{bmatrix} W_B \\ W_N \end{bmatrix} \right\} .$$

(16)

By construction, of course, $W_N^* = 0$. Using this approximation to solve (11) yields (provided $\tau_B^*$ is nonsingular, which can be ensured by construction),

$$W_B = W_B^* - [\tau_B^*]^{-1} [\tau_N^*] W_N - [\tau_B^*]^{-1} H(W_B^*, W_N^*) .$$

(17)

Taking $W_N = 0$, therefore we can define the function $G$ by the right-hand-side of (17). Newton's Method then consists of using $W_N = 0$ and the value of $W_B$ obtained from (17) to define the new value of $(W_B^*, W_N^*)$ for iterative repetition.
of the process.

Notice in (17) that each iteration of the process requires one to invert the matrix $T_B^*$. Since this is computationally burdensome, we turn our attention to the Modified Newton's Method [2,3]. In this version, the gradients (15) are not updated with every iteration: although this tends to decrease the rate of convergence, the saving in computational time per iteration more than makes up the difference. For the modified version the following recurrence relation replaces (17):

$$W_B^{(N+1)} = W_B^{(N)} - [T_B^*]^{-1} [T_N^*] W_N^{(N+1)} - [T_B^*]^{-1} H(W_B^{(N)}, W_N^{(N)}). \quad (18)$$

In practice it is advantageous to update $(T_B^*, T_N^*)$ at intervals in order to accelerate convergence, and SOLVER does this.

In (18) of course $W_N^{(N)} = 0$, and $W_N^{(N+1)} = 0$. Suppose, however, that $W_B^{(N+1)} \neq 0$, violating the nonnegativity restriction in the complementary slackness conditions. In this case we have a programming problem of the sort encountered in linear and quadratic programming; for, by interchanging certain basic and nonbasic variables we can eliminate the violation, and then proceed ahead with successive iterations of the approximation scheme. SOLVER uses Dantzig's Simplex Method for quadratic programming [1,5] to eliminate the violation, the rationale of which we present here in capsule form.

Let $W_{Bf}$ be a component of $W_B^{(N+1)}$ which is negative, $W_{Bf} < 0.0$, and then let $W_{Nf}$ be its complementary nonbasic variable. [In the terminology of the Programming Problem, if $W_{Bj}$ is a program variable then $W_{Nj}$ is the corresponding slack variable, and vice versa.] Given that $W_{Bf}$ is negative, our aim is to drive it to zero, make it a nonbasic variable, and replace it as a basic variable by its complementary variable — all the while proceeding so as to prevent any nonnegative basic variable from becoming negative. In this fashion we can systematically eliminate negative basic variables until there are none
left and the complementary slackness conditions are satisfied; thereafter, we can continue on with the iterative approximation procedure.

Rewrite (18) in the following abbreviated form:

\[ W_B = \bar{W}_B - RW_N, \tag{19} \]

where superscripts are eliminated,

\[ \bar{W}_B = W_B^{(N)} - [T_B^*]^{-1} K (W_B^{(N)}, W_N^{(N)}), \tag{20} \]

and

\[ R = [T_B^*]^{-1} T_B^*. \tag{21} \]

Assume for the moment that the matrix \( R \) is negative semi-definite; this assumption will be discussed later.

Recapping our situation: \( \bar{W}_{Bf} \) is negative, and if \( W_N = 0 \) then \( W_{Bf} \) is negative by (19), but we consider driving \( W_{Bf} \) to zero (and making it non-basic) by increasing \( W_{Nf} \) from zero (thus making it basic) without letting any nonnegative basic variables become negative. Since \( R = (r_{ij}) \) is assumed to be negative semidefinite, \( r_{ff} \leq 0 \) and we will not decrease \( W_{Bf} \) if we increase \( W_{Nf} \). Suppose that we increase \( W_{Nf} \): then the maximum amount we would want to increase it is to \( W_{Nf} = \bar{W}_{Bf}/r_{ff} \); for, at that point \( W_{Bf} \) becomes zero as desired. Increasing \( W_{Nf} \) directly to this maximum, however, ignores the possibility that one of the nonnegative basic variables might become negative; hence, in general \( W_{Nf} \) should be increased only to that value which is the smallest positive element of the set consisting of \( \bar{W}_{Bf}/r_{ff} \) and \( \bar{W}_{bi}/r_{if} \) for \( i \) running over the indices of the nonnegative basic variables. We see, therefore, that the index, say \( i^* \), of the minimal positive such element identifies the variable to be made nonbasic, and to be replaced as a basic variable by \( W_{Nf} \). The interchange of variables can, of course, be accomplished by the usual pivoting operation familiar from linear programming.

If \( i^* = f \) then we have succeeded in driving \( W_{Bf} \) to zero, as
desired, and we can now proceed to do the same thing with any other basic variables that are negative.

Suppose, on the other hand, that \( i^* \neq f \): then we must continue with our project to drive \( W_{Bf} \) to zero. For this purpose we choose to increase the value of the nonbasic variable \( W_{Ni} \) from zero, for two reasons: (1) since the former \( W_{Bi} \) was made nonbasic in the previous iteration, we know that \( W_{Ni} \) must be made basic in order to satisfy the complementary slackness conditions; (2) given the assumption that \( R \) is negative semi-definite, it can be shown [1] that \( r_{fi} \leq 0 \), so that \( W_{Bi} \) is a nondecreasing function of \( W_{Ni} \) and the latter is therefore available for driving \( W_{Bf} \) towards zero.

The method can now be outlined as follows. Assume that a floating pivot variable \( W_{Bf} \), for which \( \bar{W}_{Bf} < 0 \), is to be driven to zero. Given a nonbasic variable \( W_{Nj} \) to be increased from zero and made a basic variable, using the pivoting operation interchange \( W_{Nj} \) and that basic variable \( W_{Bi} \) for which \( i^* \) is the index of the smallest positive value in the set of ratios \( \{ \bar{W}_{Bi} / r_{ij^*} \; \text{for} \; i = f \; \text{and also} \; \text{those indices} \; i \; \text{for which} \; \bar{W}_{Bi} \geq 0 \} \). If \( i^* = f \) then terminate the procedure, since the floating pivot variable \( W_{Bf} \) has finally been driven to zero as desired; otherwise, replace \( W_{Nj} \) by \( W_{Ni} \) as the nonbasic variable to be made basic, viz., \( i^* \rightarrow j^* \) and repeat the procedure. This procedure is, of course, to be initiated with \( j^* = f \).

The justification for this procedure, and the main result proved in [1], is that given the concavity assumptions on the function \( \{ F_K(X); K = 0, 1, ..., Nc \} \) in the Programming Problem, \( R \) will always be negative semi-definite, and as a consequence that at each stage of the procedure \( r_{fj^*} \leq 0 \) so that \( W_{Bf} \) is a nondecreasing function of \( W_{Nj} \).

At this point it is easy to describe how to extend the method to handle unrestricted variables and equality constraints. If \( F_K(X) = 0.0 \) by constraint
then \( U_K \) is unrestricted in sign; hence, we need only remark on unrestricted variables. When there are unrestricted variables present, then one simply ignores them in deciding whether the restriction \( W_B \geq 0 \) in the complementary slackness conditions is satisfied, and omits them from consideration in the search for \( i^* \).

Figures 3 and 4 depict the tableau used in \( \text{SOLVER} \), both before and after the matrix \( T_B^* \) of basic columns is inverted. Observe that the matrix \( R = \left[ T_B^* \right]^{-1} T_N^* \) required for the Simplex Method occurs as the second block matrix in the inverted tableau, and that the matrix \( \left[ T_B^* \right]^{-1} \) required for the Modified Newton's Method occurs as the last block matrix. The layout of the tableau before the columns are reordered so that the columns corresponding to the basic variables occur first, is given in Figure 2.
Figure 2
Layout of the Tableau Before Reordering of the Columns

<table>
<thead>
<tr>
<th>Corresponding Variables:</th>
<th>X</th>
<th>U</th>
<th>V</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension NV</td>
<td>Q*</td>
<td>-[A*]¹</td>
<td>I</td>
<td>O</td>
</tr>
<tr>
<td>Dimension NC</td>
<td>A*</td>
<td>O</td>
<td>O</td>
<td>I</td>
</tr>
</tbody>
</table>

Dimensions: NV NC NV NC l NV NC

Note: \(N = 1\) in this situation; hence \(p(N) = p^*\) and \(b(N) = b^*\) in terms of the notational conventions adopted in Section 4, since \((x(1), u(1)) = (x^*, u^*)\) in SOLVER. In terms of the notation in Section 5, \((x, u) = \gamma\) and

\[
\begin{bmatrix}
-p(N) \\
-b(N)
\end{bmatrix} = \begin{bmatrix}
T^* & T^*
\end{bmatrix} \begin{bmatrix}
w_B(N) \\
w_N(N)
\end{bmatrix} - \eta \begin{bmatrix}
w_B(N) \\
w_N(N)
\end{bmatrix}.
\]
Figure 3
Layout of the Tableau After Reordering of the Columns
But Before Inverting the Basic Matrix

<table>
<thead>
<tr>
<th>Variables:</th>
<th>Basic</th>
<th>Nonbasic</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_B</td>
<td></td>
<td>T_N</td>
</tr>
<tr>
<td>-P(N)</td>
<td></td>
<td>B(N)</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>NV+NC</td>
</tr>
</tbody>
</table>

Dimensions:
NV+NC       NV+NC       1       NV+NC

Note: N = 1 in this situation; hence P(N) = P* and B(N) = B* in terms of the notational conventions adopted in Section 4. Since (X(l), U(l)) = (X*, U*) in SOLVER. In terms of the notation in Section 5, (X, U) = Y and

\[
\begin{pmatrix}
-P(N) \\
B(N)
\end{pmatrix}
= \begin{bmatrix}
T_B^*, T_N^*
\end{bmatrix}
\begin{bmatrix}
W_B(N) \\
W_N(N)
\end{bmatrix}
-H(W_B(N), W_N(N)).
\]
Figure 4
Layout of the Inverted Tableau

Variables:  Basic  Nonbasic

\[ R = \left[ T_B^{-1} \right] \times T_N \]

\[ \tilde{\omega}_B \]

\[ \left[ T_B^{-1} \right] \]
References


"A STRAIGHTFORWARD METHOD FOR NONLINEAR PROGRAMMING"

Robert Wilson
Assistant Professor of Business Administration, UCLA
Consultant to the RAND Corporation

The main idea is that the Kuhn-Tucker conditions for the solution of a nonlinear programming problem (maximizing a nonlinear objective function subject to nonlinear constraints, all functions concave and differentiable) can be construed as simultaneous nonlinear equations, which of course, can be solved by Newton's method. The difficulty is that the optimal basis is not known, and to find it one must employ a Simplex-type search procedure as in linear and quadratic programming. However, Newton's Method amounts merely to successively approximating the saddle function for the original problem by the saddle function for a quadratic programming problem, so Dantzig's Simplex search procedure for quadratic programming can be used in combination with Newton's Method to converge to the solution. From there on, it is a matter of optimizing the calculations (mainly by eliminating unnecessary matrix inversions); and in fact, the end result is a short, simple FORTRAN routine which is economical of both man and machine time. Special applications of the method (for which ready-made routine are available) include linear programming under uncertainty and variable-factor programming, and others will be discussed according to the audience's interests.